# 程序设计算法分析

## 信息学奥林匹克竞赛联赛知识辅导

## 上册

肖兆青

# 前　言

上世纪人类科学技术的最伟大成果非计算机莫属。自从二十世纪中叶，第一台计算机问世以来，立即引起了当代科学、技术、生产、生活和教育事业等各方面的革命。半个多世纪以来，计算机技术得到了迅猛的发展，其势可谓日新月异，她应用的领域也越见广阔，几乎没有计算机不能涉及的方面。面对这一场巨大的技术革命，计算机知识的教育已成了从小学、中学到大学的一门必不可少的课程。邓小平同志早就指示，计算机教育要从娃娃抓起，所以作为基础教育的中学，如何在普及计算机知识和技能的教学活动中，探索出一条在现代教育理论的指导下提高现代信息科技知识教学质量之路，是很值得我们研究的。

为了普及计算机知识，丰富学生课余生活，培养学生的创新思维能力，特别是对学有专长的资优学生进行个性化培养，我国每年都举办一次全国青少年信息学奥林匹克竞赛分区联赛，为全国青少年信息学奥林匹克竞赛、国际青少年信息学奥林匹克竞赛中国队组队赛、国际青少年信息学奥林匹克竞赛选拔人才。

从内容上看，这些颇具影响力和权威性的竞赛都是以程序设计为主的。有人问，计算机技术是多方面的，为什么不选基础教育课程内的文字处理、多媒体应用和网络知识，而偏偏选中程序设计呢？这是因为：

1. 程序设计与软件工具的使用不同，它要求编程者以某种高级语言为媒介，通过构造算法去解决由现实生活中抽象出来的各种问题，这些问题非一般软件工具所能解决。如果说计算机应用是“ 人脑延伸 ”的话，程序设计即为这种延伸的最高形式；
2. 程序设计对人的能力要求是多方面的。编程者不仅要熟悉计算机语言功能，而且还必须具备：

（1）扎实的数学基础和算法知识，能够对客观存在的事物及其所要解决的问题具有正确的认识和深刻的理解，包括弄清问题的需求，弄清事物的属性、行为及其彼此之间的关系；

（2）娴熟的编程技术，能够把对问题及其方法的认识描述出来，最终产生一个计算机能够理解和执行的计算机程序代码，并通过上机运行让计算机系统实现编程者解决问题的描述；

（3）较高的实践能力和创造能力，能够独立思考、提出质疑、拓延思路、洞悉规律，创造性地运用知识于不同的问题情景；

正因为程序设计能比较客观地反映人的综合素质，因此国际、国内的信息学奥林匹克竞赛都将其作为考核内容。过去是这样，现在是这样，将来恐怕亦不会随计算机技术的发展而发生改变。计算机程序设计技术将会愈来愈成为现代学生投身信息化环境，从根本上掌握现代信息技术，将来

能自由驾驭信息化时代的重要基础。计算机程序设计人才的培养，其意义不仅在收获人才，更重要的是凸现了青少年人才的培养规律，为各个领域的人才辈出提供了重要的启示。

由于程序设计的算法知识具有一定的难度要求，所以对于一些学有余力的资优学生将更具有挑战性，学习程序设计的算法知识是培养资优学生的自主学习能力和创造性思维能力的绝好平台。由于算法知识的益智性和数学味对培养资优学生的推理归纳能力、发散思维能力、构造创新能力具有事半功倍的效果，事实证明，凡是在程序设计的算法知识学习中能够自由翱翔的资优学生，他在学习其它相关领域的知识时也会感到非常的得心应手。

在对学生进行程序设计知识训练的过程中有一条非常重要，就是我们特别要注意培养学生通过自学取得独立获取知识和运用知识的能力。比如数据结构和算法的知识对学生们进一步提高程序设计能力非常关键，但由于知识的难度高、内容多，学生的水平差距可能会拉开很大，而我们的课时有可能很有限，靠教师教根本教不过来。所以我感到教师的作用更多的应该是放在精选知识点和例题上，给学生铺出一条仅用一块块不连续的大石块点缀的通路来，而这条道路的夯实和拓宽很多必须通过学生自学来解决。特别是现在计算机的知识更新很快，新的方法、新的技术层出不穷，要适应日新月异的信息时代的要求，学生掌握独立获取知识和运用知识的能力才是今后人生中能胜任终身学习的任务所必不可少的。

要培养专长就必须打破课堂教学和现有知识结构的限制，因材施教，鼓励冒尖，为资优学生们减负松绑，提供最佳的学习环境。不能要求一刀切，必要的时候要针对个别特殊情况开小灶，为资优学生们在扎实基础上的特长发挥创造条件。课堂教学最大的弊端是讲求整齐划一，它在一定程度上抑制了学生特长的发挥。从计算机程序设计知识的学习看青少年人才培养规律，启示很多。但最重要的问题还是教育观念的改变。

本人在进行了数年的《资优学生培养模式研究》课题的实践中，写下了这本关于信息学奥林匹克竞赛分区联赛中需要用到的一些程序设计算法知识的讲义，现在整理出来，抛砖引玉，供同行们批评指导。

# 第一讲　深度优先搜索

搜索是计算机程序设计竞赛中最常用的一种策略，对于那些一时难以找到规律或公式，或者根本没有规律或公式可循的问题，我们可以利用计算机高速运算的特点，用穷举策略来进行搜索。所谓穷举策略，原则地说，就是第一步只考虑问题的部分条件，根据这些条件，找到所有的满足这些部分条件的解，称之谓可行解；然后用尚未考虑的条件去一一检验那些可行解，筛去不符合条件的解，留下符合条件的解，那就是整个问题的解。

搜索可分为深度优先搜索和广度优先搜索。在深度优先搜索中，深度越大的结点越先得到扩展；而广度优先搜索则是深度越小的结点越先得到扩展。深度优先搜索的数据结构是堆栈；而广度优先搜索的数据结构是队列。下面我们首先学习深度优先搜索。

可以用深度优先搜索来处理的问题是各种各样的。如果搜索的深度已知或固定，则可用枚举法来进行；如果搜索的深度很深或不固定，则可用递归法和回溯法来进行。

## 第一节　枚举法

枚举法是一种我们十分熟悉的方法，它是用单重循环或多重循环嵌套的方法来穷举出所有可能出现的情况，然后对每种情况进行判断，找出满足条件的解来。由于用循环语句来进行穷举就必须预先知道循环变量的个数，这样才能设计循环嵌套的层数，所以它只能用于那些搜索的深度已知或固定的情况。当然，在解决一个具体的问题时，为了节省计算机的时间和空间资源，还要求我们尽可能地减少循环的层数和判断的次数，使程序的算法更加优化。

### 例 1-1-1　值班安排问题

**问题描述：**

有 A、B、C、D、E 五个人，安排值周一、周二、周三、周四、周五的班，每人值一天，共有多少种不同的安排，编程打印输出各种安排。

**算法分析：**

由于是要求五个人安排在五天中值班，每个人都可以安排在周一到周五的任意一天中值班，只能也必须安排一次，所以本题是求排列枚举的问题，只要列出这五个人在五天中的全排列情况即求得了本题的解。

如果用五重循环来对这五个人在五天中进行安排，当然可以解得本题的解，但能不能再对算法进行一下优化呢？回答是肯定的。因为在外循环中的人排定了某个值班时间后，处于内循环的人就不能重复再排在这个时间了，因为一天只能排一个人值班。这样我们只要加入一些判断条件就可

以使越到内层循环次数越少，而第五个人的安排更是不必循环了，因为这时只剩下唯一的一个时间而不容挑选了。剩下的问题是如何用数学方法来适当表述第五个人的安排时间，我们可以巧妙地利用周一到周五这五天时间的有序性，以 1、2、3、4、5 这五个数值分别代表这五天，则第五个人的安排时间就等于 15 减去前四个人的安排时间值之和，因为 15 是这五天时间值总和。

**参考程序：**

```
PROGRAM ex1_1_1(input,output);
VAR
  a,b,c,d,e : byte;
  n         : word;
BEGIN
  n := 0;
  for a := 1 to 5 do
    for b := 1 to 5 do
      if a <> b then        {排除 a、b 两人同一天值班的情况}
        for c := 1 to 5 do
          if (a <> c) and (b <> c) then
            for d := 1 to 5 do
              if (a <> d) and (b <> d) and (c <> d) then
                begin
                  e := 15-a-b-c-d;
                  {此处优化可省去一重循环，因为五天序数和为 15}
                  n := n+1;
                  writeln('No.',n:2,' :',' A : ',a,' B : ',b,
                          ' C : ',c,' D : ',d,' E : ',e);
                  if n mod 24 = 0 then readln
                end
END.
```

## 例 1-1-2　取彩球问题

**问题描述：**

在 红、黄、蓝、黑、白 五种颜色的五个彩球中，取出三个彩球的各种取法共有多少种，编程打印输出各种取法。（求组合枚举）

**算法分析：**

此题与值班安排问题的区别是这五种颜色的彩球不要求区分取出的顺序，对于取出红、黄、蓝三个球和取出黄、蓝、红三个球认为是同一种取

法，所以本题是求组合枚举的问题。

组合枚举与排列枚举的主要区别是每重循环的循环变量不需要在全部五种颜色中进行取球，因为至少要考虑两种情况下不必重复取球：一是外层循环变量已取过的颜色不必再取，二是要让出若干个位置让内层循环的变量取球。

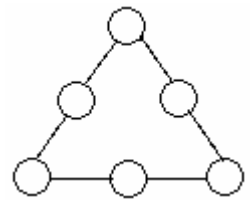
**参考程序：**

```
PROGRAM ex1_1_2(input,output);
CONST
 s:array[1..5]of string=('Red','Yellow','Blue','Black','White');
VAR
  i,j,k,n : byte;
BEGIN
  n := 0;
  for i := 1 to 3 do
  {设第 1 个位置只考虑放前三种颜色的球，因为须留出后两位置的颜色}
    for j := i+1 to 4 do {循环终值为 4 是为了留出最后 1 个球的颜色}
      for k := j+1 to 5 do begin {k 和 j 的初值要排除已取过的颜色}
        n := n+1;        {n 是不同取法的种类数计数变量}
        writeln('No.',n:2,' : ',s[i]:8,s[j]:8,s[k]:8)
      end;
  readln
END.
```

## 例 1-1-3　填数问题

**问题描述：**

在如图所示的圆圈中，不重复地填入数字 1、2、3、4、5、6。要求每条边上三个数之和都相同，编程打印输出各种填法。



**算法分析：**

此题是通过排列枚举加筛选条件来进行求解。我们可将 1 到 6 这六个数在 a、b、c、d、e、f 这六个位置进行排列枚举，再通过筛选条件：a+b+c、c+d+e、e+f+a 三者是否相等（考虑 a、c、e 为三角形的三个顶点位置）来找出符合条件的所有解。

**参考程序：**

```
PROGRAM ex1_1_3(input,output);
VAR
  s,n,a,b,c,d,e,f : integer;
BEGIN
  n := 0;
  for a := 1 to 6 do
    for b := 1 to 6 do
      if b <> a then
        for c := 1 to 6 do
          if (c <> a) and (c <> b) then
            for d := 1 to 6 do
              if (d <> a) and (d <> b) and (d <> c) then
                for e := 1 to 6 do
                  if (e <> a) and (e <> b) and (e <> c) and
                     (e <> d) then
                    begin
                      f := 21-a-b-c-d-e; {因为六数之和为 21}
                      s := a+b+c;        {考虑 a、c、e 为顶点排列}
                      if (c+d+e = s) and (a+f+e = s) then begin
                        n := n+1;        {统计解的个数}
                        writeln('No.',n:2,' : ',a:4);
                        writeln(b:10,f:4);
                        writeln(c:8,d:4,e:4); {呈三角形输出}
                        readln
                      end
                    end
END.
```

## 例 1-1-4　构造三角形问题

**问题描述：**

有长度为 3、4、5、6、7、8、9 的小木棍各一根，从中选取三根作为三角形的边长，问能构成多少种不同类型的三角形，编程打印输出各种不同类型的三角形。

**算法分析：**

此题可通过组合枚举加筛选条件来进行求解。我们可设 i、j、k 为三角形的 3 个边，并让其在 3 到 9 这 7 种规格的小木棍间以满足组合枚举规

律的方式进行循环取值。对取出的每一组组合值再用筛选条件——三角形的任意两边和应大于第三边来进行筛选，求出所有满足条件的解。

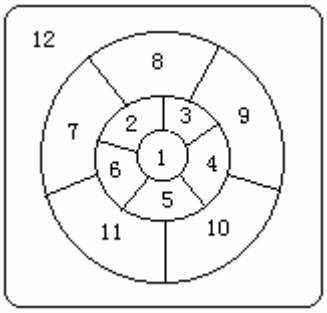
**参考程序：**

```
PROGRAM ex1_1_4(input,output);
VAR
  i,j,k,n : byte;
BEGIN
  n := 0;                        {n 用于统计解的个数}
  for i := 3 to 7 do             {留出 8、9 让后两边取值}
    for j := i+1 to 8 do         {留出 9 让最后一条边取值}
      for k := j+1 to 9 do       {k 和 j 的初值要排除已取过的边值}
        if (i+j > k) and (i+k > j) and (j+k > i) then begin
          n := n+1;  {筛选条件：三角形的任意两边和应大于第三边}
          writeln('No.',n:2,' :',i:5,j:5,k:5);
          if n mod 24 = 0 then readln
        end;
  readln
END.
```

## 例 1-1-5　地图涂色问题

**问题描述：**

下面的地图共有 12 个区域，用四种不同的颜色去着色，要求相邻区域的颜色不可相同，编程打印输出各种不同的着色方案。




**算法分析：**

此题虽然看上去很复杂，但实际上仍然是 12 个区域用 4 种颜色的排列枚举再加筛选条件就可解决的问题，而筛选条件就是相邻区域的颜色不能相同。我们可以考虑用 12 重循环代表 12 个区域，让其在 4 种颜色间进行循环，外循环从最中间的 1 开始，以后的各层循环只需考虑外循环部分的取值和当前的循环如区域相邻则取值不能相同即可。

另外，为了用循环语句输出 12 个区域的颜色值，应考虑用下标变量来做 12 层循环的循环变量，这样输出时只要将颜色数组的值输出就可以了。

**参考程序：**

```
PROGRAM ex1_1_5(input,output);
VAR
  a : array[1..12] of byte;
  n,i : byte;
BEGIN
  n := 0;
  for a[1] := 1 to 4 do
   for a[2] := 1 to 4 do
    if a[2] <> a[1] then {两个区域若相邻则不能同色，否则同色无妨}
     for a[3] := 1 to 4 do
      if (a[3] <> a[2]) and (a[3] <> a[1]) then
       for a[4] := 1 to 4 do
        if (a[4] <> a[3]) and (a[4] <> a[1]) then
         for a[5] := 1 to 4 do
          if (a[5] <> a[4]) and (a[5] <> a[1]) then
           for a[6] := 1 to 4 do
            if (a[6] <> a[5]) and (a[6] <> a[1]) and
               (a[6] <> a[2]) then
             for a[7] := 1 to 4 do
              if (a[7] <> a[6]) and (a[7] <> a[2]) then
               for a[8] := 1 to 4 do
                if (a[8] <> a[7]) and (a[8] <> a[2]) and
                   (a[8] <> a[3]) then
                 for a[9] := 1 to 4 do
                  if (a[9] <> a[8]) and (a[9] <> a[3]) and
                     (a[9] <> a[4]) then
                   for a[10] := 1 to 4 do
                    if (a[10] <> a[9]) and (a[10] <> a[4]) and
                       (a[10] <> a[5]) then
                     for a[11] := 1 to 4 do
                      if (a[11] <> a[10]) and (a[11] <> a[5]) and
                        (a[11] <> a[6]) and (a[11] <> a[7]) then
                       for a[12] := 1 to 4 do
```

```
                    if (a[12] <> a[7]) and (a[12] <> a[8]) and
                       (a[12] <>a[9]) and (a[12] <> a[10]) and
                       (a[12] <> a[11]) then begin
                      n := n+1;
                      write('No.',n,' : ');
                      for i := 1 to 12 do
                        write('(',i,':',a[i],')');
                      writeln;
                      if n mod 24 = 0 then readln
                    end
END.
```

# 第二节　递归法

对于搜索的深度很深或深度不固定的情况，则无法用枚举的方法来设置循环嵌套的层数，这时可以考虑用递归法来完成搜索任务。递归是一种常用算法，它是搜索的另一种实现方式。如果在算法设计中采用一个函数或过程直接或间接地调用它自身来解决问题的方法，则称该方法为递归算法。递归算法必须要设计好一个或若干个确定的递归终止条件。

下面我们先从 4 个求排列和组合的例子来分析递归算法的解题过程和解题方法，掌握递归算法的解题思路和原理，熟悉用递归算法求解问题的一般规律。

## 例 1-2-1　有重复值的全排列问题

**问题描述：**

由键盘输入一个正整数 n，在 1 到 n 之间作 n 重嵌套的循环穷举并打印输出，即输出 1 到 n 中有重复值的 n 个数字的全排列。

例如当 n=2 时的输出应为：

1、1；　1、2；　2、1；　2、2　共 4 种。

**算法分析：**

我们可以用一个数组 a : array[1..n] of byte；来存放每轮等待输出的 n 个值，剩下的问题就是如何将每轮的值准确无误地赋给 a 数组，并在打印输出过一轮的值之后又能准确地变换为下一轮的一组值。

我们可以从 1 开始将递归层数参量带入递归过程，而在递归子程序中可设计一重 1 到 n 的 i 循环来控制同一深度的 a 数组值的变化，并在该循环中即将递归深入一层。下面我们来分析一下程序的执行过程：

由于递归是按深度进行的，所以对应一个相同的 i，首先是进行逐层递归，而各递归层的 a 数组元素是赋的相同的 i 值。直至达到递归终止条件（深度 s > n）时为止，打印输出第一轮 a 数组的值（全是 1）。终止递归后返回上一层（n 层），循环变量 i 的值加 1 后再递归，又会满足递归终止条件而打印输出第二轮 a 数组的值……直至将 1 到 n 的变化完成（这时打印输出了 n 轮 a 数组的值，每组值只是 a[n]的值不同，其他都是 1）。再递归终止返回时由于 n 层的 i 循环已经结束，所以再继续返回至 n-1 层。将 n-1 层的 i 值加 1 后再递归，……以此类推，直至第一层的 i 循环完成 1 到 n 的变化后，并各层的 a 数组元素值都为 n 时，打印输出最后一轮全为 n 值的 a 数组值。输出完成后的返回将是连续返回 n 层，回到主程序结束程序的运行。

**参考程序：**

```
PROGRAM ex1_2_1(input,output);
```

```
VAR
  a : array[1..9] of byte;
  n : byte;
PROCEDURE sub(s : byte);
  var
    i,j : byte;          {s > n 是递归终止条件：表示 s > n 时终止递归，
  begin                  打印 a 数组内容，打印后并未结束，而是退回递归
    if s > n             前的 i 循环的下一个 i 值……继续递归、打印}
      then begin
             for j := 1 to n do  write(a[j]:2);  writeln
           end
      else for i := 1 to n do
             begin
               a[s] := i;    {递归前 a 数组 s 元素赋值 i，
               sub(s+1)       且递归一次，赋值一次}
             end
  end;
BEGIN
  repeat             {输入 n 的值，范围控制在 1 到 9 之间}
    write('N = ');    readln(n)
  until n in [1..9];
  sub(1);  readln   {递归初始值从 1 开始}
END.
```

从以上程序我们可以看到，递归程序的特点是它既有递进过程，又有回归过程，而且在任何一个深度时，它的所有变量和参数的值由于采用的是传值方式，所以都在各层保存着，这是因为 Pascal 系统在递归递进的过程中会将同一变量或参数在不同深度的值，都依次压入系统的堆栈。由于堆栈结构先进后出的特点，所以深度浅的数据先进栈，深度深的数据后进栈；在递归的回归过程中，由于堆栈结构后进先出的特点，深度深的数据先出栈，深度浅的数据后出栈。这就给我们在递归过程中能对数据进行有条不紊的处理提供了可靠的保证。

Pascal 系统为我们提供了一种监视窗口的功能，我们可通过跟踪并观察变量和参数 a、s、i 变化的方法来弄清楚以上程序的机理及来龙去脉。

## 例 1-2-2　无重复值的全排列问题

**问题描述：**

由键盘输入一个正整数 n，在 1 到 n 之间进行 n 重嵌套的，每重 1 到 n 的排列穷举，且每个数组元素所赋的值，必须是各不相同的，即求出无重复值的全排列。

例如 n=2 时的输出为：

1、2； 2、1 共 2 种。

**算法分析：**

此题与上一题的区别是每轮的输出数组元素中不能有相同的值。为了排除相同值的出现，我们可以设置一个标志数组 b : array[1..9] of boolean；其值为 true 时可用，为 false 时不可用。

初始时，先对 b 数组全置 true；进入递归过程后，首先判断 b[i]的值是否为 true,若为 true 才能给 a 数组元素赋 i 值,否则继续判下一个 b[i]的情况；在对 a 数组的某个元素赋了一个 i 值后，立即将其对应的 b[i]值置 false,以保证在以后的递进过程中不赋重复的 i 值，……直至满足递归终止条件后打印输出满足题意的一组解；另外要注意，在退出递归的回归过程中，每退出一层都要立即将该层原来设置的用过的 b[i]标志恢复为 true,以便于在再一次递归进入时可将 i 用于下一组输出的 a 数组元素的赋值之中。

**参考程序：**

```
PROGRAM ex1_2_2(input,output);
VAR      {设置一个标志数组 b，true 为可用，用过置为 false}
  a : array[1..9] of byte;
  b : array[1..9] of boolean;
  n : byte;
PROCEDURE sub(s : byte);
  var
    i,j : byte;
  begin    {实质上在递归过程中 i 是外循环，s 是内循环，通过 j 输出}
    if s > n
      then begin
             for j := 1 to n do  write(a[j]:2);  writeln
           end
      else for i := 1 to n do
             if b[i] then {条件成立表示 i 可用，否则判下一个 i 值}
               begin
                 a[s] := i;
                 b[i] := false;  {对赋过的 i 其 b[i]标志置 false}
                 sub(s+1);   {每层赋了一个值后即递归进入下一层}
```

```
                b[i] := true {退出后标志恢复，进入返回层 i 循环}
              end
  end;
BEGIN
  fillchar(b,sizeof(b),true);
  {标准过程：表示将 b 数组范围的所有数组元素赋初值 true，表示可用}
  repeat  write('N = ');  readln(n)  until n in [1..9];
  sub(1);  readln
END.
```

## 例 1-2-3　无重复值的部分元素排列问题

**问题描述：**

打印输出从 n 个元素中选取 m 个元素的各种排列（1≤n、m≤9）。即求出从 n 个元素中选取 m 个元素的无重复值的部分元素排列。

例如当 n=3、m=2 时，其输出应为：

1、2； 1、3； 2、1； 2、3； 3、1； 3、2　共 6 种。

**算法分析：**

此题与上一题的区别仅仅是从 n 个元素中取值的个数不同，上一题是全取 n 个，而本题是只取其中的 m 个，所以我们只要将上一题的递归终止条件 s > n 改为 s > m 即可，当然 j 循环的打印输出终值也要相应改为 m。另外在主程序的输入部分要增加判断 m 不能超出 n 的范围的语句，以保证输入的 n 和 m 的值能够正确运行。

**参考程序：**

```
PROGRAM ex1_2_3(input,output);
VAR
  a   : array[1..9] of byte;
  b   : array[1..9] of boolean;
  n,m : byte;
PROCEDURE sub(s : byte);
  var
    i,j : byte;
  begin
    if s > m        {与上题的区别仅在将两处的 n 改为 m ，
      then begin     这样外循环终值为 n ,内循环终值为 m }
             for j := 1 to m do  write(a[j]:2);
             writeln
           end
```

```
      else for i := 1 to n do     {取值范围仍然是 1 到 n}
             if b[i] then
               begin
                 a[s] := i;
                 b[i] := false;  sub(s+1);  b[i] := true
               end
  end;
BEGIN
  fillchar(b,sizeof(b),true);
  repeat            {输入部分要判断 m 不能超出 n 的范围}
    write('N = ');  readln(n);
    write('M = ');  readln(m)
  until (n in [1..9]) and (m <= n);
  sub(1);  readln
END.
```

## 例 1-2-4　部分元素的组合问题

**问题描述：**

打印输出从 n 个元素中选取 m 个元素的各种组合（1≤n、m≤9）。

例如当 n=3、m=2 时，输出结果应为：

1、2； 1、3； 2、3　共 3 种。

**算法分析：**

此题与上一题的区别是只要求出不同的组合即可，与各数字的排列顺序无关，即相同元素的不同顺序排列认为是相同的。

为了保证输出的每一组的 m 个元素各不相同，就要求递归时不同深度的 a 数组取值必须各不相同，我们可以让递归每进入一层，递归过程的参量就增加 1 来达到此目的，这就是每次递归都是以 sub（s+1）进入的原因。但是本题的组合求解还有一个要求，就是相同元素的不同顺序排列不能出现，这就要求每一递归层的 i 初始值也要比前一层的 i 初始值递进，这样才能保证前面输出过的元素不再重复输出。这里我们可以把 i 的初始值设置为 a[s-1]+1，即在上一层的 a 数组元素值上再加 1，当然上题使用的标志数组 b 这里也不需要了。

**参考程序：**

```
PROGRAM ex1_2_4(input,output);
VAR
  a   : array[0..9] of byte;    {求组合不需标志数组}
  n,m : byte;
```

```
PROCEDURE sub(s : byte);
  var
    i,j : byte;
  begin
    if s > m
      then begin
             for j := 1 to m do  write(a[j]:2);  writeln
           end
      else for i := a[s-1]+1 to n do
{请与题 ex1_2_1、题 ex1_2_3 比较，可以看到循环变量 i 在不同深度的初始值是不同的，它保证了 i 的递进，使得每一行输出的数字各不相同}
             begin  a[s] := i;  sub(s+1)  end
  end;
BEGIN
  repeat
    write('N = ');  readln(n);
    write('M = ');  readln(m)
  until (n in [1..9]) and (m <= n);
  sub(1);  readln
END.
```

上面我们通过几个求排列和组合问题的简单例子讨论了用递归算法求解问题的基本规律，下面我们再举几个实际搜索应用的例子，来进一步巩固递归算法的知识。

## 例 1-2-5　皇后问题

**问题描述：**

在 n×n 的国际象棋的棋盘上，放 n 个皇后，要做到各皇后相互之间不能攻击，问共有多少种不同的放置方法，要求打印输出各种放置方法。（2≤n≤9，“相互之间不能攻击”表示放置时每行每列每一条对角线都不能超过一个皇后）

**算法分析：**

本题可以用递归进行全排列搜索，再用筛选条件挑选出满足题意的解来。为了使程序结构清晰易读，我们设置了 print 过程用于输出满足条件的每一组解；设置了 check 函数用于判断在某个位置能否放置新的皇后。

本题的题意是所有的皇后都不能同行、同列或同斜线，由于我们将递归的深度参量 s 与行编号同值，所以放置的皇后就不可能同行，这样就不

须判断皇后是否同行，只要判是否同列和同斜线就行了。

另外，我们将数组 a 的下标值定为行编号的值，而数组各元素的值定为列编号的值，这样就可以用一个数组的存储容量同时解决了行和列的存储问题，节约了存储空间。

**参考程序：**

```
PROGRAM ex1_2_5(input,output);
VAR
  a   : array[1..9] of 1..9;
  n,t : word;
PROCEDURE print;
  var
    j : byte;
  begin
    t := t+1;  write('No.',t,' : ');   {统计不同放置方法数}
    for j := 1 to n do  write('(',j,',',a[j],') ');
    writeln;     {输出皇后的放置位置，j 表示行，a[j]表示列}
    if t mod 24 = 0 then readln    {换屏暂停}
  end;
FUNCTION check(s,i : byte) : boolean;
  var   {本函数用于判断 s 行 i 列位置能否放新的皇后，能放为 true}
    j : byte;
  begin
    check := false;        {行循环变量 j 只要在当前行 s 前面的各行
    for j := 1 to s-1 do   进行循环比较即可，因 s 行后还未放皇后}
      if (i = a[j]) or (abs(s-j) = abs(i-a[j])) then exit;
    check := true      {条件 1:同列；条件 2:同斜线(行差与列差绝对
  end;                  值相等表示了以位置 s,i 为交点的两条斜线)}
PROCEDURE sub(s : byte);
  var
    i : byte;
  begin
    if s > n           {递归终止条件。满足一次打印一组解}
      then print       {输出用子程序处理}
      else for i := 1 to n do   {在 s 行的各列查找能放皇后的位置}
             if check(s,i) then
               begin    {check 子程序用于判断 s,i 位置能否放皇后}
                 a[s] := i;  { a[s]记录新皇后的列位置，s 是行位置}
```

```
                sub(s+1)    {递归搜索下一行}
              end
  end;
BEGIN
  repeat
    write('N = ');  readln(n) {n 的取值范围限定在 2 到 9 之间}
  until n in [2..9];
  t := 0;                     {放置方法计数器置初值}
  sub(1);                     {递归搜索从第 1 行开始}
  readln
END.
```

## 例 1-2-6　字母组合问题

**问题描述：**

由键盘输入正整数 M、N、C，输出从第 M 个小写英文字母开始，顺序地取 N 个小写英文字符，在 N 个小写英文字母中，取 C 个字母的所有组合。

**算法分析：**

这是一个典型的组合求解题，与题 ex1_2_4 相比仅仅是增加了英文字母的 ASCII 值和数值之间的转换问题。当然我们也可以用字母的有序值设置循环变量的值，这样虽然可以省去了英文字母的 ASCII 值和数值之间的转换，但是递归程序中的数值计算方面却会变得复杂得多。权衡利弊，还是采用数值方法来控制递归过程，到输出时再进行一次 ASCII 值和数值之间的转换来达到解题的目的。

**参考程序：**

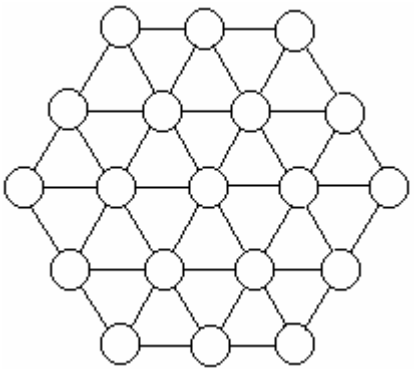
```
PROGRAM ex1_2_6(input,output);
VAR
  a     : array[1..26] of byte;
  m,n,c : byte;
  t     : longint;
PROCEDURE print;  {输出过程在找到解的基础上将其转换为之母输出}
  var
    j : byte;
  begin
    t := t+1;
    write('No.',t,' : (');
    for j := 1 to c-1 do            {此 95 再加上 m 和 a[j]的起始
      write(chr(95+m+a[j]),',');    值 1 正好 97,为'a'的 ASCII 码}
```

```
    writeln(chr(95+m+a[c]),')');
    if t mod 24 = 0 then readln
  end;
PROCEDURE sub(s : byte);
  var
    i : byte;
  begin
    if s > c  {递归终止条件为取得了 c 个数}
      then print
      else for i := a[s-1]+1 to n do{取数从上一层的取值加 1 开始}
             begin
               a[s] := i;  sub(s+1) {取数后即递归进入下一层}
             end
  end;
BEGIN
  t := 0;
  repeat  {输入限定条件是 m+n 不能超出字母范围，组合数在取数范围}
    write('Input M,N,C : ');  readln(m,n,c)
  until (m + n <= 27) and (c <= n);
  sub(1);  readln
END.
```

## 例 1-2-7　填数问题

**问题描述：**

在如下图所示的 19 个圆圈中，不重复地填入数字 1 至 19，要使每条直线上的几个数字之和都相等，请输出一种填写方法。



**算法分析：**

本题初看时可能会觉得一下子难以找出规律，但从数学角度分析一下

就可以看出每条直线的数字和应为 38。因为我们可以先求出 1 到 19 的数字和为 190，然后再找出无重复元素且包含所有 19 个数的 5 条直线（比如水平的 5 条直线），就可以知道每条直线的和是 190/5 = 38 ，题中把它设为常量 k。

接下来我们就可以采用递归求无重复值排列的算法来求解，在求解过程中再加入筛选条件（即是否满足每条直线上的数字和都为 38）就可以求出解来。由于直线数有 15 条之多，所以要尽量避免重复判断，本题中在每一个位置试放数字解题的过程中，因为后面的位置还未放数字，所以只要判断与前面位置能构成完整直线的线上数字之和是否为 38 就行了。如满足条件就进入下一层递归，试解下一位置的值；如不满足条件则表示该位置的取值无效，重新取下一个值再试；如果该位置与前面的位置不能构成一条完整直线，则取数有效，进入下一层递归，试解下一位置的值。

为了保证取过的数字不再被重复取用，我们可以设置一个标志数组。只要该数组元素的下标值（代表 1 到 19 这 19 个数之一）已被取用，就将该数组元素的值置为 false，以后位置再试解取数时必须先判断该数对应的标志数组元素是否为 true，为 true 才能取用，否则就不能用。当然对试用无效的数别忘了恢复其数组标值后再选其它数，以免该数被漏用。

当逐个找出 19 位置的解以后，就可以输出结果了。我们设置了 print 过程来完成输出任务。为了输出结果能比较形象地表示图示的 19 个位置，程序中用输出 5 行数据，且每个数据以一定的场宽值来控制其输出位置的方法来输出，结果形象又直观。

根据题意只要求输出一种填写方法,所以输出完毕即用 halt 命令强制程序结束，不然可以输出更多组解来。

**参考程序：**

```
PROGRAM ex1_2_7(input,output);
CONST
  k = 38;      {此值由 19 个数的总和除以 5 行得到,为每直线的和值}
VAR
  a : array[1..19] of 1..19;  {用于存放 19 个位置所置的数}
  b : array[1..19] of boolean;{标志数组:1 到 19 中取过值为 false}
PROCEDURE print;  {按图示格式打印输出 19 个数}
  begin
    writeln(a[1]:6,a[2]:4,a[3]:4);
    writeln(a[4]:4,a[5]:4,a[6]:4,a[7]:4);
    writeln(a[8]:2,a[9]:4,a[10]:4,a[11]:4,a[12]:4);
    writeln(a[13]:4,a[14]:4,a[15]:4,a[16]:4);
    writeln(a[17]:6,a[18]:4,a[19]:4);
```

```
    readln;  halt          {根据题意只要输出一种结果即可退出}
  end;
PROCEDURE sub(s : byte);
  var
    i : byte;
  begin
    if s > 19
      then print   {满足条件表示找到了一组解，即执行 print 过程}
      else for i := 1 to 19 do     {在 1 到 19 中循环试解}
             if b[i] then    {标志数组元素允许才能试解}
               begin
                 a[s] := i;  {i 被置数后其标志数组值相应置 false}
                 b[i] := false;
                 case s of   {以下判断只需考虑小于 s 的位置情况}
                   3  : if a[1]+a[2]+a[3] = k then sub(s+1);
                   7  : if a[4]+a[5]+a[6]+a[7] = k then sub(s+1);
                   8  : if a[1]+a[4]+a[8] = k then sub(s+1);
                   12 : if (a[8]+a[9]+a[10]+a[11]+a[12] = k) and
                        (a[3]+a[7]+a[12] = k) then sub(s+1);
                   13 : if a[2]+a[5]+a[9]+a[13] = k then sub(s+1);
                   16 : if (a[13]+a[14]+a[15]+a[16] = k) and
                        (a[2]+a[6]+a[11]+a[16] = k) then sub(s+1);
                   17 : if (a[8]+a[13]+a[17] = k) and (a[3]+a[6]
                        +a[10]+a[14]+a[17] = k) then sub(s+1);
                   18 : if (a[4]+a[9]+a[14]+a[18] = k) and (a[7]
                        +a[11]+a[15]+a[18] = k) then sub(s+1);
                   19 : if (a[17]+a[18]+a[19] = k) and
                        (a[1]+a[5]+a[10]+a[15]+a[19] = k) and
                        (a[12]+a[16]+a[19] = k) then sub(s+1);
                   else sub(s+1) {s 不是上述值则取值有效直接递归}
                 end;
                 b[i] := true
               end  {s 取值不满足条件则恢复标志位，试解下一值}
  end;
BEGIN
  fillchar(b,sizeof(b),true);    {标志数组置初值 true}
  sub(1)
```

```
END.
```

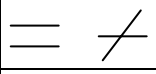
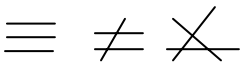
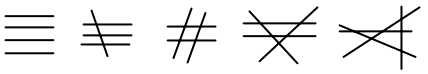
通过以上的一些例子我们已经基本掌握了递归算法的一般规律，但是在解决实际问题的使用中我们要解决的问题往往会比前面所述的问题要困难得多、复杂得多，很多问题我们并不能一下子就能看出可以用递归方法来进行搜索求解，而是要求我们先要对问题进行数学模型的分析，通过分析找出能使用递归算法的数学模型，甚至还要想方设法地去构建出一个合适的递归模型，这样才能运用递归算法来解决问题。这时候，能不能想出一个可行的递归算法模型就会成为能不能求解该问题的关键。特别是在信息学程序设计竞赛中，一个问题的算法模型建立得正确与否、优劣与否在很大程度上决定了程序设计的水平。很多同学在事后去看别人编写的程序会感觉并不是很难，后悔当时自己为什么就没有想到采用这种方法。实际上这里主要的原因还是对递归算法的理解不够深透，运用不够熟练，构造算法模型的本领还不强。如何进一步提高自己的编程水平呢？只有不断通过对各种问题特别是疑难问题的分析、求解，经常思考和实践如何对一些比较复杂的实际问题去构造一种我们熟悉的、简化的、符合计算机程序设计算法的数学模型，才能不断提高我们的程序设计水平。

为了使大家能进一步提高递归算法的运用能力，下面我们选取的例子将增加数学模型分析方面的难度，并且在算法分析中也不再详细讨论递归过程的一般方法和处理技巧，而是将重点放到对一个看似复杂的具体问题如何去进行数学模型的分析，如何从中找出我们熟悉的符合递归算法的模型来，就是说要想方设法地去构造出我们需要的符合递归算法的模型。

## 例 1-2-8　平面直线的交点问题

**问题描述：**

在平面上有 n（0 < n <= 25）条无限长直线，且无三线共点，问这些直线能组成多少种不同的交点数。例如：

| n（直线数） | p（可能的交点数） | 图形示例 |
|---|---|---|
| 1 | 0 | |
| 2 | 0，1 |  |
| 3 | 0，2，3 |  |
| 4 | 0，3，4，5，6 |  |

**算法分析：**

① 交点的最大数目出现在 n 条直线全部不平行且无三线交于一点的情况，其值为 n（n－1）/2，所以可设置数组：

a：array[0.. n（n－1）/2] of boolean

来表示某交点数是否可能，true 为可能，false 为不可能；

② 以 n = 6 为例，我们作如下分析：

a. 当 n 条直线全部平行时，交点的个数为 0，即 a[0]=true；

b. 当有 1 条直线不平行时，有 f（1）种情况：
基础交点数为（n － 1）*1 = 5，再在此交点数上分别加上 1 条直线能产生的交点数情况（1 种 0 个交点的情况），即 5+0=5，这时只有 a[5]=true，a[1]=a[2]=a[3]=a[4]=false；

c. 当有 2 条直线不平行时，有 f（2）种情况：
基础交点数为（n-2）*2 =8，再在此交点数上分别加上 2 条直线能产生的交点数情况（2 种情况，分别为 0 和 1 个交点），即 8+0=8 和 8+1=9，这时 a[8]=a[9]=true；

d. 当有 3 条直线不平行时，有 f（3）种情况：
基础交点数为（n-3）*3=9，再在此交点数上分别加上 3 条直线能产生的交点数情况（3 种情况，分别为 0、2、3 个交点），即 9+0=9、9+2=11、9+3=12，这时 a[9]=a[11]=a[12]=true；
……

e. 讨论一般情况，设有 r 条直线不平行，则此时有 f（r）种交点数产生的情况：其交点数为（n － r）* r 再分别加上 f（r）情况时的交点数；

③ 可见，要求得 f（n）就必须先求得 f（n-1）、f（n-2）、……、f（1），显然，用递归方法可方便地解得本题的答案。

**参考程序：**

```
PROGRAM ex1_2_8(input,output);
CONST
  max=300;  {交点的最大数目为 n（n－1）/2}
VAR
  n,k,i,s:integer;
  a      :array[0..max] of boolean;
PROCEDURE try(m,j:integer);
  var
    p:integer;
  begin
    if m>0       {递归终止条件是无平行线，此时的交点数 j 是一种解}
```

```
      then for p:=1 to m do
             try(m-p,j+p*(m-p))
      else a[j]:=true
  end;
BEGIN
  repeat  {输入直线数。但不能超出交点数的最大值范围}
    write('n=');  readln(n);  k:=n*(n-1) div 2
  until k<=max;
  for i:=0 to k do  a[i]:=false;  {所有可能交点数的记录数组置初值}
  try(n,0);          {从 n 条直线全平行（交点数为 0）的情况搜索起}
  s:=0;  writeln(' No. ','     Dots');
  for i:=0 to k do          {输出不同交点数的种数和对应的交点数}
    if a[i] then begin  inc(s);  writeln('[',s:3,']',i:8)  end;
  readln
END.
```

{p 从直线数 m 中循环试解有 1 到 m 条直线不平行的情况，递归第 1 参量是平行线数，第 2 参量是能产生的交点数。j 从 0 开始每次增加 p*(m-p)}

## 例 1-2-9　移动棋子问题

**问题描述：**

有 2n 个棋子（50＞n≥4）排成一行，开始位置为白子全部在左边，黑子全部在右边,最右边是 2 个用于交换移动棋子的空位,如图所示(n=4)：

○○○○●●●●__ __

移动棋子的规则是：每次必须同时移动相邻的两个棋子，颜色不限，可以左移也可以右移到空位上去，但不能调换两个棋子的左右位置。每次移动必须跳过若干个棋子（不能平移），最后要求能移成黑白相间的一行棋子，2 个空位在最左边。如图所示（n=4）：

__ __○●○●○●○●

**算法分析：**

①当 n=4 时我们可以通过以下 5 步移动成功：

初始时：　○○○○●●●●__ __

第 1 步：　○○○__ __●●●○●

第 2 步：　○○○●○●●__ __●

第 3 步：　○__ __●○●●○○●

第 4 步：　○●○●○●__ __○●

第 5 步：　__ __○●○●○●○●

②当 n=5 时我们要找规律，使 n=5 的情况变化为 n=4 的情况，以利于递归处理：

初始时：　○○○○○●●●●●__ __

第 1 步： ○○○○__ __●●●●○●

第 2 步： ○○○○●●●●__ __○●

③以此类推，要求 n=m 时的移动方法，可以先移动 2 次，然后求 n=m-1 的移动方法，……，直至 n=4 时给出上述的 5 步解法作为递归终止条件。

④程序中设置了 c 数组用于存放当前的棋子状态，其存储空间为 2n+2。并且每改变一次数组存储状态即输出打印一次，这样可以清楚地看到棋子的移动变化过程。

⑤递归处理的任务由 mv 递归过程完成。mvsp 过程的任务是协助 mv 过程完成数组中具体的棋子移动赋值的任务。

**参考程序：**

```
PROGRAM ex1_2_9(input,output);
CONST
  max=100;  {最大值为：49*2+2}
VAR
  i,n,st,sp:integer;  {st 是移动步骤计数器，sp 是交换空位的指针}
  c        :array[1..max] of char;    {棋子状态数组}
PROCEDURE print;      {输出当前的移动步骤数和当前的棋子状态}
  begin
    write('step ',st,' : ');
    for i:=1 to 2*n+2 do  write(c[i]);  writeln;
    inc(st)           {输出当前棋子状态后移动步骤计数器加 1}
  end;
PROCEDURE init(n:integer);
  begin
    st:=0;      {移动步骤计数器置初值}
    sp:=2*n+1;  {因为 2 个空位一起移动，所以只需指向第 1 个空位}
    for i:=1 to n do    {棋子状态数组前 n 位置白，接着 n 位置黑}
      begin  c[i]:='0';  c[n+i]:='@'  end;
    c[2*n+1]:='_';  c[2*n+2]:='_';  {留出最右边的 2 个交换空位}
    print                           {输出 step 0 的初始状态}
  end;
PROCEDURE mvsp(k:integer);
  begin
    for i:=0 to 1 do     {完成第 k 位和 k+1 位向空位移动的任务}
      begin  c[sp+i]:=c[k+i];  c[k+i]:='_'  end;
    sp:=k;  print        {置新空位指针后输出当前棋子状态}
  end;
```

```
PROCEDURE mv(n:integer);
  var
    k:integer;
  begin
    if n=4   {满足递归终止条件后按 n=4 的分析方法移动}
      then begin
             mvsp(4);     {将 4、5 位置的棋子移到 9、10 位值}
             mvsp(8);     {将 8、9 位置的棋子移到 4、5 位值}
             mvsp(2);     {将 2、3 位置的棋子移到 8、9 位值}
             mvsp(7);     {将 7、8 位置的棋子移到 2、3 位值}
             mvsp(1)      {将 1、2 位置的棋子移到 7、8 位值}
           end
      else begin
             mvsp(n);  {将 n、n+1 位置的棋子移到 2n+1、2n+2 位置}
             mvsp(2*n-1);{将 2n-1、2n 位置的棋子移回 n、n+1 位值}
             mv(n-1)      {n 降阶处理一次后进入 n-1 阶的递归处理}
           end
  end;
BEGIN
  repeat           {在限定的范围内输入 n 的值}
    write('Input n(50>n>3): ');  readln(n)
  until (n>3) and (n<50);
  init(n);  { 将 c 数组初始化为原始状态，称为 step 0}
  mv(n);    {用递归过程完成 2n 个棋子的移动}
  readln
END.
```

## 例 1-2-10　猴子选大王问题

**问题描述：**

有 n 只猴子选大王，先从头至尾 1 … p 报数，报到 p 的猴子退出；报至尾后，再从尾至头继续报数（注意：如果报到尾时不是 p，则反过来报数仍然从尾开始，即尾部的这个猴子将连续被报 2 个数），仍然是报到 p 的猴子退出；报到头后，再从头继续，……；直至剩下最后 1 个猴子，即为选出的大王。

要求输入 n，p（n≤100，p≤n）以后，能正确输出大王的初始编号。

**算法分析：**

①整个报数过程是一个有规律递减的过程，且有明确的终止条件，所以能

够运用递归算法来解决本问题；

②编程中要注意控制正反方向报数的不同数据处理方法，以得到正确求解。

③猴子数 n 因不断递减，可设置为传址方式的参量或全局变量形式。

**参考程序：**

```
PROGRAM ex1_2_10(input,output);
TYPE
  arr=array[1..100] of 0..1;
VAR
  h          :arr; {猴子是否在列的记录数组，1 为在列，0 为已出列}
  i,m,n,p,f,s:byte;
PROCEDURE num(var n:byte);
  begin
    if n>1 then  {递归终止条件为只剩 1 个猴子}
      begin
        case i mod 2 of        {判断是正方向还是反方向报数}
          1:for f:=1 to m do              {正方向报数处理}
              if h[f]<>0 then      {只对在列的猴子进行处理}
                begin
                  inc(s);               {报数变量递增}
                  if s=p then begin     {报到 p 作以下处理}
                                s:=0;   {报数计数重新开始}
                                dec(n); {猴子数减 1}
                                h[f]:=0 {该猴子清除出数组}
                              end
                end;
          0:for f:=m downto 1 do   {反方向报数处理与正方向类似}
              if h[f]<>0 then
                begin
                  inc(s);
                  if s=p then
                    begin  s:=0;  dec(n);  h[f]:=0  end
                end
        end;
        inc(i);    {i 是报数方向计数器，奇数为正向，偶数为反向}
        num(n)     {用新的在列猴子数进入新的递归}
      end
  end;
```

```
BEGIN
  write('Input n,p:');  readln(n,p);
  for i:=1 to n do  h[i]:=1;       {猴子是否在列的记录数组置初值}
  i:=1;                {报数方向计数器置初值，表示正方向开始报数}
  m:=n;  {m 记录猴子总数，用于正反方向对全数组扫描，因为 n 将变小}
  s:=0;                {报数计数器置初值}
  num(n);              {开始以猴子总数进入递归}
  i:=1;  while h[i]=0 do inc(i);     {找出在列的唯一一个猴子}
  writeln('The leader is:',i)        {即选出的大王}
END.
```

## 例 1-2-11　表达式去括号问题

**问题描述：**

设有如右形式的代数表达式：　(a+b) * (c-d)

去掉括号后可得到：(a+b) * (c-d) = a * c – a * d + b * c – b * d

题意约定：

运算符为：+、-、*、/ 四种，且无单目运算；

运算数为：小写的英文字母。

问题要求：

接受输入一个符合题意的代数表达式，能正确输出去掉括号后的对应表达式。

**算法分析：**

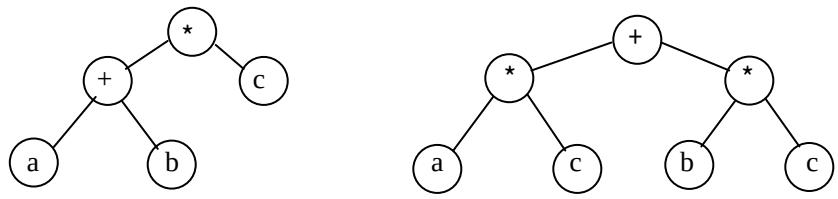
①本题是对代数式求双目运算，所以采用二叉树的数据结构（指针结构）较为方便。如：

```
Tnode     =^Tnodedata;
Tnodedata=record
            ch :char;      {数据域：用于存放运算数或运算符}
            l,r:Tnode      {指针域：分别指向左右子树}
          end;
```

②我们先考虑只有 +、* 两种运算符的情况（以中序遍历的方式表述）：

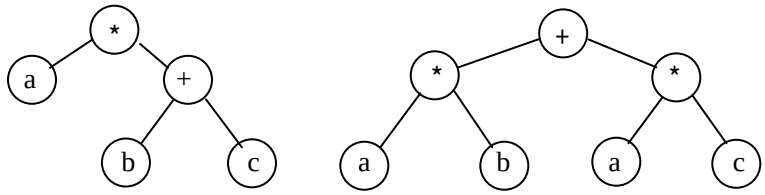
情况一：

(a + b) * c　⟶　a * c + b * c



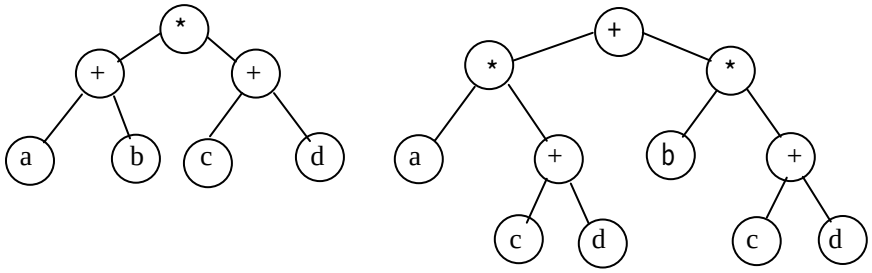
情况二：

a * (b + c) ⟶ a * b + a * c

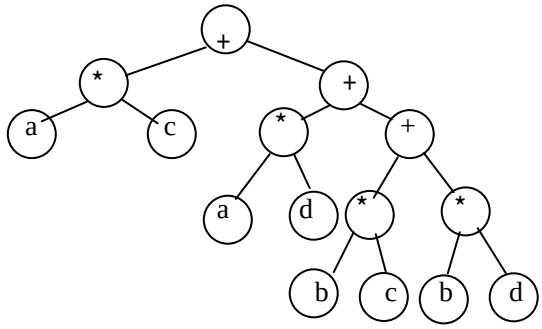



情况三：

(a + b) * (c + d) ⟶ a * (c + d) + b * (c + d)




⟶ a * c + a * d + b * c + b * d




综合上面三种特例可以看出，当子树的根是 * 号，且左或右子树是 + 号时，需要进行二叉树的重新整理，直至 * 号的子树无 + 号为止。

③递归处理的过程为：

if 根=‘ * ’and 左子树=‘ + ’then

计算出‘ + ’的左子树去括号表达式 A$ 和‘ + ’的右子树去括号表达式 B$；

使二叉树变成：A$ *(‘ * ’号的右子树) + B$ * (‘ * ’号的右子树)

if 根=‘ * ’and 右子树=‘ + ’then

计算出‘ * ’号的左子树去括号表达式为 A$；

计算出‘ + ’号的左、右子树去括号表达式为 B$、C$；
使二叉树变成：A$ * B$ + A$ * C$

④当考虑存在 / 号的情况时：
/ 号与 * 号情况不同之处是：
当 / 号的右子树是带括号的 +、- 运算时将不能去括号（如 c/（a+b））；
当 / 号的右子树是带括号的 *、/ 运算时其 *、/ 号应相应改变为 /、* 运算号。

⑤当考虑存在 - 号的情况时：
- 号与 + 号情况不同之处是，要考虑 - 号后去括号时括号内的加减号要进行变号处理。

⑥编程中首先要解决的问题是如何将原始表达式化成一棵二叉树。参考程序中我们是采用了三层函数嵌套的方法来完成此任务的。我们首先将表达式中的字符分成三类：一类是运算数，按题意只能是小写字母，放在最内层，作为树叶子处理，无下级内容；第二类是运算符中的乘除号，放在中层，以乘除号为界，前面内容为左子树，后面内容为右子树；第三类是运算符中的加减号，放在外层，以加减号为界，前面内容为左子树，后面内容为右子树。三层函数调用的结果除了运算数是树叶子外，其他的运算符将是按照表达式中运算符出现的前后顺序来安排二叉树的结点层次的。对于表达式中的括号我们不在结点的数据域中放置，而是采取二叉树中序遍历的形式隐含表述的，也就是说中序遍历的顺序就是运算的顺序。

⑦接下来的问题是将原始表达式二叉树转换成去括号后的表达式二叉树。我们根据本题算法分析 2 至 5 所述的变换原则进行了处理，实际上是将原始表达式二叉树中的乘除号结点层次降阶到加减号之下，也就是说除了除号的右子树为加减号的情况外不再存在加减号结点层次在乘除号以下的情况。

⑧由于本题的解法采用了指针结构，所以在编程中要仔细考虑二叉树各结点的数据域和指针域的赋值及其结点间的联系。因为树空间不同于数组空间，数组空间是线性固定的，而指针结构的树空间却是动态的层次结构，所以就要求我们的头脑中必须对每一步的树型结构的变化有一个全面的正确的认识，否则就会出现差错。

**参考程序：**

```
PROGRAM ex1_2_11(input,output);
TYPE
  Tnode    =^Tnodedata;
  Tnodedata=record
              ch :char;      {数据域：用于存放运算数或运算符}
```

```
            l,r:Tnode      {指针域：分别指向左右子树}
          end;
VAR
  root:Tnode;              {根结点}
  s   :string;             {存放输入表达式}
  ptr :integer;            {表达式中字符位置指针}
  bb  :boolean;            {用于判断表达式树的去括号任务是否完成}

FUNCTION newnode(data:char; lc,rc:Tnode):Tnode; {建立一个新结点}
  var
    nd:Tnode;
  begin
    new(nd);
    with nd^ do {根据调用本函数的参量定义新结点的数据域、指针域}
      begin  ch:=data;  l:=lc;  r:=rc  end;
    newnode:=nd            {将建立的新结点作为函数值输出}
  end;
FUNCTION expression:Tnode;       {外层函数定义}
  var       {通过函数三层嵌套定义、递归调用产生一棵原始表达式树}
    expr:Tnode;
  function expr2:Tnode;          {中层函数定义}
    var
      tm:Tnode;
    function expr3:Tnode;        {内层函数定义}
      begin
        case s[ptr] of
          'a'..'z':begin    {对运算数（小写字母）作为树叶子处理}
                     inc(ptr);{树叶结点数据域赋字母值，指针域空}
                     expr3:=newnode(s[ptr-1],nil,nil)
                   end;
          '('     :begin                    {对左括号作跳过处理}
                     inc(ptr);           {括号内的内容作为新一轮
                     expr3:=expression;  表达式重新进行递归调用}
                     inc(ptr)                {返回时再跳过右括号}
                   end
        end
      end;
```

```
    begin
      tm:=expr3;  {进入中层函数前先调用内层函数，排除字母和括号}
      while s[ptr] in ['*','/'] do              {对乘除号的处理}
        begin
          inc(ptr);       {左、右子树分别指向乘除号前、后的内容}
          tm:=newnode(s[ptr-1],tm,expr3)  {右子树是函数参数调用}
        end;
      expr2:=tm             {函数值作为外层函数中加减号的右子树}
    end;
  begin
    expr:=expr2;  {进入外层函数前先调用中层函数，排除乘除号……}
    while s[ptr] in ['+','-'] do                {对加减号的处理}
      begin
        inc(ptr);         {左、右子树分别指向加减号前、后的内容}
        expr:=newnode(s[ptr-1],expr,expr2){右子树是函数参数调用}
      end;
    expression:=expr  {建好的原始表达式树的根结点作为函数值输出}
  end;
FUNCTION copytree(n:Tnode):Tnode;
  var                 {本函数的作用是复制一棵以 n 为根结点的子树}
    ct:Tnode;
  begin
    copytree:=nil;  if n=nil then exit;     {n 结点为空则不复制}
    new(ct);
    ct^.ch:=n^.ch;          {复制结点数据域}
    ct^.l:=copytree(n^.l);  {递归复制左子树}
    ct^.r:=copytree(n^.r);  {递归复制右子树}
    copytree:=ct            {将 n 为子树根结点作为函数值输出}
  end;
PROCEDURE change(n:Tnode);  {本函数是去括号处理的主要功能程序段}
  var
    ta,tb:Tnode;            {暂存新产生的左、右子树结点}
    yc   :char;             {数据域内容暂存变量}
  begin
    if n=nil then exit;     {递归终止条件。满足条件返回树叶结点}
    change(n^.l);           {先递归搜索左子树}
    change(n^.r);           {再递归搜索右子树}
```

```
  if n^.ch in ['*','/'] then     {对乘除号的处理}
    begin
      if n^.l^.ch in ['+','-'] then
        begin                  {对乘除号的左子树为加减号的处理}
          yc   :=n^.l^.ch;           {左子树的加减号数据暂存}
          ta   :=newnode(n^.ch,nil,nil);  {用乘除号建立新左树}
          ta^.l:=n^.l^.l;      {新结点的左指针指向加减号的左域}
          ta^.r:=copytree(n^.r);  {其右指针指向乘除号的右子树}
          tb   :=newnode(n^.ch,nil,nil);  {用乘除号再建新右树}
          tb^.l:=n^.l^.r;      {新结点的左指针指向加减号的右域}
          tb^.r:=n^.r;         {新结点的右指针指向乘除号的右域}
          n^.ch:=yc;         {原乘除号结点数据域赋暂存的加减号}
          n^.l :=ta;                 {新树的左指针指向新左树}
          n^.r :=tb;                 {新树的右指针指向新右树}
          bb   :=true;  exit    {去括号操作标志变量赋值后退出}
        end;
      if (n^.r^.ch in ['+','-']) and (n^.ch='*') then
        begin   {对乘号（除号情况不同）的右子树为加减号的处理}
          yc   :=n^.r^.ch;           {右子树的加减号数据暂存}
          ta   :=newnode(n^.ch,nil,nil);    {用乘号建立新左树}
          ta^.l:=n^.l;           {新结点的左指针指向乘号的左域}
          ta^.r:=n^.r^.l;      {新结点的右指针指向加减号的左域}
          tb   :=newnode(n^.ch,nil,nil);    {用乘号再建新右树}
          tb^.l:=copytree(n^.l);    {其左指针指向乘号的左子树}
          tb^.r:=n^.r^.r;           {其右指针指向加减号的左域}
          n^.ch:=yc;           {原乘号结点数据域赋暂存的加减号}
          n^.l :=ta;                 {新树的左指针指向新左树}
          n^.r :=tb;                 {新树的右指针指向新右树}
          bb   :=true;  exit    {去括号操作标志变量赋值后退出}
        end
    end;
  if (n^.ch='/') and (n^.r^.ch in ['*','/']) then
    begin {对除号的右子树为乘除号的处理（加减号则不能去括号）}
      yc:=n^.r^.ch;              {除号的右子树的乘除号数据暂存}
      case yc of
        '*' : yc:='/';   '/' : yc:='*'              {变号处理}
      end;
```

新左子树 新右子树 新树 新左子树 新右子树 新树

```
                ta:=newnode('/',nil,nil);              {用除号建立新左树}
新左子树        ta^.l:=n^.l;                {新结点的左指针指向除号的左域}
                ta^.r:=n^.r^.l; {新结点的右指针指向原右子树乘除号的左域}
                n^.ch:=yc;              {原除号结点数据域赋变号后的乘除号}
新树            n^.l:=ta;         {新树的左指针指向以除号为根结点的新左树}
                n^.r:=n^.r^.r;     {新树的右指针指向原右子树乘除号的右域}
                bb  :=true;  exit          {去括号操作标志变量赋值后退出}
              end;                    {乘除号处理结束，以下是非乘除号情况}
          if (n^.ch='-') and (n^.r^.ch in ['+','-']) then
            begin                            {减号的右子树是加减号的处理}
              yc:=n^.r^.ch;               {减号的右子树的加减号数据暂存}
              case yc of
                '-' : yc:='+';   '+' : yc:='-'                {变号处理}
              end;
                ta:=newnode('-',nil,nil);              {用减号建立新左树}
新左子树        ta^.l:=n^.l;                {新结点的左指针指向减号的左域}
                ta^.r:=n^.r^.l;           {新结点的右指针指向加减号的左域}
                n^.ch:=yc;              {原减号结点数据域赋变号后的加减号}
新树            n^.l :=ta;        {新树的左指针指向以减号为根结点的新左树}
                n^.r :=n^.r^.r;           {新树的右指针指向原加减号的右域}
                bb   :=true;  exit         {去括号操作标志变量赋值后退出}
              end
        end;                  {除了以上几种情况外的其他情况都不必作处理}
    PROCEDURE changetree;
      begin {反复对表达式树进行去括号检查和转换，直至无转换对象为止}
        repeat  bb:=false;  change(root)  until not(bb)
      end;
    PROCEDURE showtree(n:Tnode; ku:boolean);  {输出去括号后的表达式}
      var                        {第 2 参量用于判断是否要增加括号的输出}
        nu:boolean;
      begin
        if n=nil then exit;             {递归终止条件}
        if ku then write('(');          {当第 2 参量为 true 输出左括号}
        nu:=false;                      {进入递归搜索前第 2 参量置 false}
        showtree(n^.l,nu);              {递归搜索左子树}
        write(n^.ch);                   {输出结点数据}
        if (n^.ch = '/') and (n^.r^.ch in ['+','-']) then nu:=true;
```

```
    showtree(n^.r,nu);     {在进入右子树搜索前先判断是否要加括号}
    if ku then write(')')          {当第 2 参量为 true 输出左括号}
  end;
PROCEDURE distree(n:Tnode);          {本过程用于释放树的存储空间}
  begin
    if n=nil then exit;              {递归终止条件}
    distree(n^.l);  distree(n^.r);  {递归搜索左右子树}
    dispose(n)                       {释放结点空间}
  end;
BEGIN
  write('Input expression:');  readln(s);
  ptr:=1;              {输入要去括号的表达式后指针指向第 1 个字符}
  write(s,' = ');      {先输出原表达式}
  new(root);           {产生一个根结点 root}
  root:=expression;    {将原表达式化成一棵以 root 为根的二叉树}
  changetree;          {将原表达式树改变成一棵去除括号后的树}
  showtree(root,false);  writeln;    {输出去除括号后的表达式}
  distree(root);  readln             {回收二叉树结点空间}
END.
```

## 例 1-2-12　推广的哈夫曼编码问题

**问题描述：**

哈夫曼编码是众所周知的，即对给出的一个字符串进行 0，1 编码，要求总的编码长度为最短。

推广的哈夫曼编码是对给出的字符串采用 0，1，2，……，p 进行编码，使总的编码长度为最短。

本题的要求是对给出的字符串采用 0，1，2 三种数值进行编码，并满足：

① 编码是唯一的，即不同的字母（限小写英文字母）有不同的编码；

② 使总的编码长度为最短；

③ 程序能做到：输入一串小写英文字母能正确输出符合条件的编码；再输入一串符合条件的编码能还原成原来的字符串。

**算法分析：**

(1) 编码的唯一性问题：当 p≥1 时，编码的唯一性是成立的。

(2) 怎样才能做到编码的长度为最短呢？

① 编码的方法与字符出现的频数有关：即某个字符出现的频数多时，其编码长度应尽可能少，所以我们首先要对每个字符出现的频数进行排

序。例如，输入字符串为： abcaabcdeffedfaaxyytfb

统计排序的结果为：

a（5 次）f（4 次）b（3 次）c（2 次）d（2 次）e（2 次）y（2 次）x（1 次）t（1 次）

② 为了便于归并，要根据不同字符的数目进行补足，补足的方法是：

- 补足字符的出现频数应设为 0；
- 补足字符后的结果应使归并过程的最后一轮归并对象数正好为 3；
- 可以推算，当 n=3+2k 时，不用补 0 频数字符，否则应补 1 个 0 频数字符。

③ 归并过程示例（以上述输入字符串为例）：

- 因为有 9 种字符，所以不需补足操作，首先归并频数最少的 3 个字符：

t（1）、x（1）、c（2）、d（2）、e（2）、y（2）、b（3）、f（4）、a（5）

④

重新排序后得：

d（2）、e（2）、y（2）、b（3）、（t（1）x（1）c（2））、f（4）、a（5）

④

- 再对频数最少的 3 个字符进行第 2 次归并：

d（2）、e（2）、y（2）、b（3）、（t（1）x（1）c（2））、f（4）、a（5）

⑥ ④

再重新排序后得：

b（3）、（t（1）x（1）c（2））、f（4）、a（5）、（d（2）e（2）y（2））

④ ⑥

- 继续对频数最少的 3 个字符（已归并的字符组视为 1 个字符）进行第 3 次归并：

b（3）、（t（1）x（1）c（2））、f（4）、a（5）、（d（2）e（2）y（2））

④ ⑥

⑪

再重新排序后得：

a（5）、（d（2）e（2）y（2））、b（3）、（t（1）x（1）c（2））、f（4）

6　　4

11

- 再对最后 3 个对象进行最后 1 次归并：

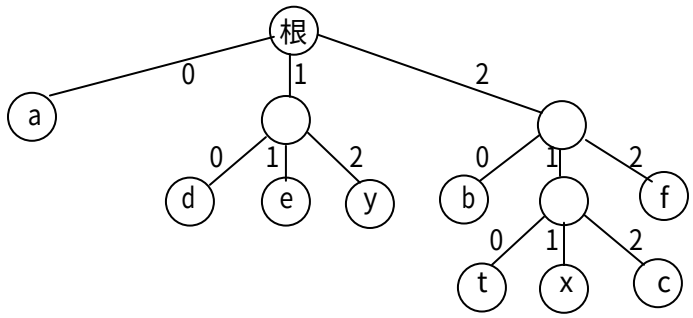
a（5）、（d（2）e（2）y（2））、b（3）、（t（1）x（1）c（2））、f（4）

6　　4

11

根

④ 从上面的归并过程可以得到 1 棵哈夫曼树：




⑤ 我们设左子树为 0，中子树为 1，右子树为 2，则可以得到各字母的哈夫曼编码为：

a__0、b__20、c__212、d__10、e__11、f__22、t__210、x__211、y__12

⑥ 当本程序输入 abcaabcdeffedfaaxyytfb 一串字符后，程序应输出：

0 20 212 0 0 20 212 10 11 22 22 11 10 22 0 0 211 12 12 210 22 20

这样一串哈夫曼编码（其中每个编码间的空格是为了便于大家分析编码而设，实际应用中是没有空格的）；

(3) 题目还要求实现哈夫曼编码的反编码功能，即当本程序输入：

0 20 212 0 0 20 212 10 11 22 22 11 10 22 0 0 211 12 12 210 22 20

这样一串哈夫曼编码后（其中每个编码间的空格同样是为便于大家分

析编码而设，实际不必输入），则应能输出还原的字符串：
abcaabcdeffedfaaxyytfb。

**参考程序：**

```
PROGRAM ex1_2_12(input,output);
TYPE
  Tnode    =^Tnodedata;
  Tnodedata=record
              c    :char;   {记录字母名称}
              n    :byte;   {记录该字母出现的频数}
              l,m,r:Tnode   {这是一棵三叉树，所以设置 3 个指针}
            end;
VAR                         {采用数组指针结构存放小写字母和空格}
  fore     :array[1..27] of Tnode;
  s,pt,code:string;
  t,i,j    :integer;
PROCEDURE del(a:integer);                    {删除指定的一个结点}
  begin
    for j:=a+1 to t+1 do  fore[j-1]:=fore[j];   {各结点前移覆盖}
    fore[t+1]^.c:='!'           {数组尾部不用结点处加入终止标记}
  end;
PROCEDURE sort(n:integer);{本过程是典型的加标志变量的选择排序法}
  var
    p:Tnode;
    k:integer;
  begin
    for i:=1 to n-1 do
      begin
        k:=i; {标志变量的设置是为了保证同频字母能按字母顺序排列}
        for j:=i+1 to n do  if fore[k]^.n>fore[j]^.n then k:=j;
        if k<>i then  {设置标志变量保证了一轮最多只交换一次数据}
          begin  p:=fore[i];  fore[i]:=fore[k];  fore[k]:=p  end
      end
  end;
PROCEDURE init;
  begin
    write('Enter a string:');  readln(s);{输入待编码的小写字母串}
    for i:=1 to 27 do               {数组指针结构的各结点初始化}
```

```
    begin
      new(fore[i]);                                {建立一个结点}
      fore[i]^.c:=chr(i-1+ord('a'));       {数据域存放字母名称}
      fore[i]^.n:=0;                     {频数为 0 ;各指针域为空}
      fore[i]^.l:=nil;  fore[i]^.m:=nil;  fore[i]^.r:=nil
    end;
  for i:=1 to length(s) do               {扫描检查输入的字母串}
    begin
      if not (s[i] in ['a'..'z']) then  {有非小写字母出错退出}
        begin  writeln('Error in string!');  halt  end;
      inc(fore[ord(s[i])-ord('a')+1]^.n)  {统计各字母出现频数}
    end;
  i:=1;  t:=26;                {字母种数统计计数器从 26 开始递减}
  while (fore[i]^.c<>'!') and (i<=26) do
    if fore[i]^.n=0   {去掉未用的字母结点，统计使用的字母种数}
      then begin  del(i);  dec(t)  end
      else inc(i);    {i 不一定能扫描到 26，要随结点删除而减少}
  if not(odd(t)) then {字母种数不是 3+2k（奇数）要补 0 频率字符}
    begin
      inc(t);  fore[t]^.n:=0;  fore[t]^.c:=' '    {用空格补入}
    end;
  sort(t)          {对使用的所有字符包括补用的空格进行升序排序}
end;
PROCEDURE work;     {建立一棵推广的哈夫曼树}
  var
    p:Tnode;
  begin
    while t>1 do{t=1 时表示归并结束，产生最后根结点，哈夫曼树形成}
      begin
        p:=fore[1];          {暂存第 1 个频数最小的结点内容}
        new(fore[1]);
        new(fore[1]^.l);
        new(fore[1]^.m);
        new(fore[1]^.r);
        fore[1]^.l:=p;
        fore[1]^.m:=fore[2];
        fore[1]^.r:=fore[3];
```

{产生 4 个新结点，用于存放归并后的根和 3 个被归并的频数最小的结点}

{将 3 个频数最小的结点按从小到大的顺序分别存放到对应的左、中、右 3 棵子树中}

```
        fore[1]^.n:=p^.n+fore[2]^.n+fore[3]^.n;
        fore[1]^.c:=' ';{新归并的结点名称为空格,频数为3结点之和}
        del(3);  del(2);  }{删除序号2、3的结点（子树中已保存），在
        dec(t,2);         }{待排序列中保留1号结点（已改为归并根）}
        sort(t)          {对新序列重新进行升序排序}
      end
  end;
PROCEDURE pr(p:Tnode; ch:char);
  begin     {本过程的作用是在哈夫曼树中找出指定字母的哈夫曼编码}
    if p<>nil then         {递归终止条件是一直递归搜索到树叶结点}
      if p^.c=ch
        then code:=pt     {将搜索到的哈夫曼编码记录到code中送出}
        else begin
               pt:=pt+'0';  pr(p^.l,ch);{pt加0后递归搜索左子树}
               delete(pt,length(pt),1);       {返回时pt再删去0}
               pt:=pt+'1';  pr(p^.m,ch);{pt加1后递归搜索中子树}
               delete(pt,length(pt),1);       {返回时pt再删去1}
               pt:=pt+'2';  pr(p^.r,ch);{pt加2后递归搜索右子树}
               delete(pt,length(pt),1)        {返回时pt再删去2}
             end
  end;
PROCEDURE print;  {输出字母串的哈夫曼编码}
  var
    p:Tnode;
  begin
    p:=fore[1];              {p是哈夫曼树的根结点}
    for i:= 1 to length(s) do {逐个扫描原输入的字母串，查哈夫曼码}
      begin
        pt:='';        {记录对应字母的哈夫曼码变量设置初值(空串)}
        pr(p,s[i]); {从哈夫曼树的根结点开始递归搜索对应字母编码}
        write(code,' ')   {输出搜索到的哈夫曼编码，再加空格分隔}
      end;
    writeln
  end;
PROCEDURE rework;          {本过程可将一串哈夫曼编码还原成字母串}
  var
    p:Tnode;
```

```
  procedure dd;                {本过程用于判断是否搜索到树叶结点}
    begin
      if (p^.l=nil) and (p^.m=nil) and (p^.r=nil) then
        begin
          write(p^.c);  {搜索到树叶结点时即输出该结点数据域字母}
          p:=fore[1]              {输出字母后指针重新指向根结点}
        end
    end;
  begin
    write('Enter a code:');  readln(s);       {输入哈夫曼编码串}
    i:=1;  p:=fore[1];
    repeat    {按哈夫曼码 0、1、2 的指引去逐层搜索左、中、右子树}
      case s[i] of
        '0':begin  p:=p^.l;  dd  end;
        '1':begin  p:=p^.m;  dd  end;
        '2':begin  p:=p^.r;  dd  end
      end;
      inc(i)             {取下一个哈夫曼码}
    until i>length(s);   {取完所有哈夫曼码后循环终止}
    writeln
  end;
BEGIN
  init;    {完成字母串输入；统计频数；0 频率字符加补；排序等工作}
  work;    {建立一棵哈夫曼树}
  print;   {输出哈夫曼编码}
  rework   {将输入的哈夫曼编码还原成字母串}
END.
```

# 第三节　回溯法

回溯法是一种非递归形式的深度优先搜索，它是用一个数组和两个计数器来实现的。一个计数器对数组元素的下标进行计数；另一个计数器对数组元素所赋的值进行计数。它相当于对每个数组元素的赋值做一重或多重循环。下面我们仍旧用递归搜索法中举过的 4 个例子来进行说明（见第二节的例题 ex1_2_1、 ex1_2_2、 ex1_2_3 、ex1_2_4 关于排列和组合的问题），请大家通过比较加深理解，掌握回溯算法的搜索方法和原理。

## 例 1-3-1　有重复值的全排列问题

**问题描述：**

由键盘输入一个正整数 n，在 1 到 n 之间作 n 重嵌套的循环穷举并打印输出，即输出 1 到 n 中有重复值的 n 个数字的全排列。

例如当 n=2 时的输出应为：

1、1； 1、2； 2、1； 2、2 共 4 种。

**算法分析：**

回溯搜索的方法是利用一个数组和两个计数器来进行工作的。在本题中我们设置了一个数组 a : array[0..9] of byte；对动态的数据进行存放，并设置 i 计数器用以控制 a 数组的下标变化，又设置了 j 计数器用以对 a 数组的各元素进行赋值。

本程序的运行过程如下：

程序开始运行后先输入一个 n 值，范围在 1 到 9 之间；然后进入 i 计数器控制的外循环，i 在初值 0 基础上加 1；接着又进入 j 计数器控制的内循环，j 也在初值 0 的基础上加 1；

接下来的一个条件判断非常关键：当 j<=0 时搜索深度不断增加，且一边增加一边给 a[i]赋 j 值，由于接下来的语句 if i = n then print else break 的作用，只要 i < n 就会每次对 a[i]赋 j 值后在 break 的作用下立即中止内循环。这样外循环变量 i 的值不断增加，内循环变量 j 却由于每次都是重新开始循环而重置初值，直到 i = n 时执行 print 过程，数组 a 的所有元素值都是 1，这就是输出的第 1 行解；

执行 print 过程结束，返回内循环，这时由于 i 在内循环中不改变值，始终满足 i = n，而 j 却在内循环中不断加 1，所以 a[n]的值不断赋新值，且改变一次，执行 print 过程一次，这样就完成了输出 n 组解，这 n 组解的内容除了 a[n]的值从 1 到 n 变化外,a[1]到 a[n-1]各元素的值始终未改变过，一直为 1；

当 j > n 时内循环执行 else 部分，i := i-1，i 变量开始回溯，j 变量重新赋 a[i] 值，这时的 a[i]是刚刚输出了一轮变化值的 a[n]的前 1 位

数组元素 a[n-1]，也就是说 j 要在 a[n-1]的基础上重新循环计数，且 j 加 1 后由于 j 又满足了 j <= n 的条件，内循环又执行 then 部分，a[i]赋 j 值，实际上达到了 a[n-1]增加 1 的目的，为后面的 a[n]位重新循环变化输出创造了条件；

由于回溯后的 i 不等于 n，执行 break 命令，退出内循环，让 i 加 1 后再重起内循环，j 的计数又从 1 开始……；总之，a[n]的值循环一轮，a[n-1]的值增加 1，……，a[n-1]的值循环一轮，a[n-2] 的值增加 1，……；

由于回溯的条件是 j > n，所以一般情况下 i 回溯 1 位，就会又进入新的深度搜索，只有在 j 读取的前一数组元素的值为 n 的情况下才会继续回溯，因为这时执行 j := j+1 又能满足 j > n 的回溯条件；……；

如果所有的数组元素的值都已经计数增加到 n 了，也就是说所有的解都已经搜索完毕了，这时的回溯过程将能够从 a[n]位一直回溯到 a[1]位；再继续回溯 i 就等于 0 了，满足了程序的终止条件。

以上我们详细分析了回溯算法的原理，并详细讨论了数组变量值、计数器变量值变化的过程，这对我们真正掌握回溯算法，能应用回溯算法举一反三解决实际问题是非常重要的。编程的具体形式可以千变万化，但对算法的原理却要真正领会，这才是学好算法知识的关键。

**参考程序：**

```
PROCEDURE ex1_3_1(input,output);
VAR
  a       : array[0..9] of byte;
  i,j,k,n : byte;
PROCEDURE print;
  begin
    for k := 1 to n do   write(a[k]:3);
    readln                        {这里既有停顿作用，又有换行作用}
  end;
BEGIN
  repeat  write('N = ');  readln(n)  until n in [1..9];
  i := 0;                 {i 是外循环控制变量，用于控制数组下标}
  repeat
    i := i+1;  j := 0;       {j 是内循环控制变量，并赋给数组为值}
    repeat
      j := j+1;              { i<n 时：break 的作用是跳出 j 循环，使 j
      if j <= n              值重起,i 值加 1；i=n 时：执行打印，执行后
        then begin           由于 i 未变，j 值增加，故打印 j 循环一
               a[i] := j;    圈..... 直至 j>n 时，转 else 部分}
```

```
            if i = n then print else break
          end
      else begin
            i := i-1;                                    {i 回溯}
            if i = 0 then halt;  {回溯到 i=0 表示数组搜索结束}
            j := a[i]  {j 赋前一列的起始值，继续 j 循环..... }
          end
  until false
until false
END.
```

## 例 1-3-2　无重复值的全排列问题

**问题描述：**

由键盘输入一个正整数 n，在 1 到 n 之间进行 n 重嵌套的，每重 1 到 n 的排列穷举，且每个数组元素所赋的值，必须是各不相同的，即求出无重复值的全排列。

例如 n=2 时的输出为：

1、2；　2、1　　共 2 种。

**算法分析：**

本题与上题的区别是各数组元素中不允许赋有相同的值，这样在输出时就不会出现重复值。所以我们在程序中专门增加了一个 check 函数，用以判断将要给数组元素赋的 j 值是否前面已经赋过，如已经赋过则函数值为 false，表示不能赋此 j 值，继续试解下一个 j 值；若前面没有赋过则函数值为 true，表示可以赋此 j 值。

check 函数的搜索只要从 1 到 i-1 间进行，因为当前的赋值元素是 a[i]，i 以后的数组值还待定。

**参考程序：**

```
PROGRAM ex1_3_2(input,output);
VAR
  a       : array[1..9] of byte;
  i,j,k,n : byte;
FUNCTION check : boolean;
  begin              {本函数对 a[i]之前的各数组元素进行搜索，看
    check := false;    是否存在 j，存在为 false，不存在为 true}
    for k := 1 to i-1 do   if j = a[k] then  exit;
    check := true
  end;
```

```
PROCEDURE print;
  begin
    for k := 1 to n do  write(a[k]:3);  readln
  end;
BEGIN
  repeat
    write('N = ');  readln(n)
  until n in [1..9];
  i := 0;
  repeat
    i := i+1;  j := 0;
    repeat
      j := j+1;
      if j <= n          {与上题相比,这里仅仅增加了 check 函数,用
        then begin        于排除在 a 数组各元素中有相同的 j 值出现}
               if check then begin
                               a[i] := j;
                               if i = n then print else break
                             end
             end
        else begin
               i := i-1;
               if i = 0 then halt;
               j := a[i]
             end
    until false
  until false
END.
```

## 例 1-3-3　无重复值的部分元素排列问题

**问题描述：**

打印输出从 n 个元素中选取 m 个元素的各种排列（1≤n、m≤9）。即求出从 n 个元素中选取 m 个元素的无重复值的部分元素排列。

例如当 n=3、m=2 时，其输出应为：

1、2； 1、3； 2、1； 2、3； 3、1； 3、2　共 6 种。

**算法分析：**

本题与上题的区别是输出排列要求选取的元素个数不同，上题是全取

n 个元素输出，而本题是只从 n 个元素中选取部分元素输出，所以外循环变量 i 只要以 m 为终值就行了，实际上数组 a 的 n 个元素也只需使用 m 个元素就行了。

为了解决上述问题，我们只要把内循环中控制执行输出过程 print 的条件 i = n 改为 i = m，同时输出子程序 print 过程中的打印循环终值也改为 m 就行了。这样，原来要全取 n 个不同 j 值才能输出，现在只要选取了 m 个不同 j 值就可以输出了。

当然，内循环变量 j 的变化范围仍然在 1 到 n 之间，因为 m 个元素的取值范围并没有改变。

**参考程序：**

```
PROGRAM ex1_3_3(input,output);
VAR
  a         : array[1..9] of byte;
  i,j,k,n,m : byte;
FUNCTION check : boolean;
  begin
    check := false;
    for k := 1 to i-1 do  if j = a[k] then exit;
    check := true
  end;
PROCEDURE print;
  begin
    for k := 1 to m do  write(a[k]:3);
    writeln                              {控制一行、打印 m 个数据}
  end;
BEGIN
  repeat
    write('N = ');  readln(n);
    write('M = ');  readln(m)
  until (n in [1..9]) and (m <= n);
  i := 0;
  repeat
    i := i+1;  j := 0;
    repeat
      j := j+1;
      if j <= n
        then begin
```

```
            if check then begin
                          a[i] := j;
                          if i = m then print else break
                        end        {每行的元素个数限定为 m}
          end
      else begin
             i := i-1;
             if i = 0 then halt;
             j := a[i]
           end
  until false
until false
END.
```

## 例 1-3-4　部分元素的组合问题

**问题描述：**

打印输出从 n 个元素中选取 m 个元素的各种组合（1≤n、m≤9）。

例如当 n=3、m=2 时，输出结果应为：

1、2； 1、3； 2、3　共 3 种。

**算法分析：**

从 n 个元素中选取 m 个元素的组合与排列的区别是选出的 m 个元素不要考虑排列的顺序，即不同顺序排列的 m 个元素是无区别的。

为了避免不同顺序排列的 m 个元素重复出现，我们在递归算法中是让不同递归深度的循环变量 i 的初始取值总是在前一层的数组元素取值上加 1 递进，这就保证了后面的数组元素取值总是比前面的数组元素取值大，也就避免了不同顺序排列的 m 个元素重复出现的问题。在回溯算法中我们能不能也可以采用这个思路呢？回答是肯定的。但不同的是，这里要控制的是内循环变量 j 而不是外循环变量 i，因为在回溯算法中给数组元素赋值的是内循环变量 j。

怎样才能实现上述思路呢？我们可以在内循环进入之前对内循环变量 j 赋初值 0 的语句上动脑筋。j 赋初值 0 的目的是为了让 j 的候选值每次从 1 开始，这是求排列的解法所要求的。而求组合则要求数组中后面的元素赋值一定要大于前面的元素赋值，这样 j 的初值就不能为 0，而应该是前一项数组元素所赋的值，即 j 应该等于 a[i-1]。这时 j 的候选值就会在 a[i-1]的基础上加 1 后再进行试解，用这种方法求得的解一定是后面的数组元素值大于前面的数组元素值，就不会出现不同顺序排列的 m 个元素重复出现的问题，这也就求出了本题要求的各种组合解。

**参考程序：**

```
PROGRAM ex1_3_4(input,output);
VAR
  a         : array[1..9] of byte;
  i,j,k,n,m : byte;
PROCEDURE print;
  begin
    for k := 1 to m do  write(a[k]:3);
    writeln
  end;
BEGIN
  repeat
    write('N = ');  readln(n);
    write('M = ');  readln(m)
  until (n in [1..9]) and (m <= n);
  i := 0;
  repeat
    i := i+1;
    j := a[i-1];           {j 的初值决定了内循环的取值始终大于外
    repeat                  循环的值，这就保证了组合效果的形成}
      j := j+1;
      if j <= n then begin
                       a[i] := j;
                       if i = m then print else break
                     end
                else begin
                       i := i-1;
                       if i = 0 then halt;
                       j := a[i]
                     end
    until false
  until false
END.
```

以上我们用回溯算法讨论了 4 个关于排列和组合问题的搜索方法，初步理解了回溯算法中一个数组和两个计数器通过双重循环控制怎样一步步“深入”进行搜索，又怎样一步步“回溯”，并且巧妙地、有控制地、恰到

好处地将“深入”和“回溯”交替运行，解出了我们需要的一组组的解。通过解题分析，我们可以初步掌握回溯算法的基本规律。下面我们就运用这些规律来解决一些实际问题，通过回溯算法知识的应用，进一步巩固学到的回溯算法知识。

## 例 1-3-5　跳马问题（1）

**问题描述：**

有一个 m×n 的中国象棋棋盘（3≤n、m≤30）。在其左下角有一只马，马走日字，并且规定马只许往右跳，不许往左跳，请你设计一个程序，由计算机寻找并打印出马从左下角到右上角的所有可能的跳行路线。

程序输入：

棋盘的行、列宽度值 m、n；

程序输出：

以坐标表示的马的所有跳行路径（左下角坐标 0、0，右上角坐标 m、n）。

**算法分析：**

跳马问题是一种典型的搜索问题。粗略一看马的跳法众多，且各个不同位置能采用的跳法也不同，好像一时难以找出规律。但我们如果将每一步可行的的跳点位置作为搜索的深度，再将某一位置的所有允许跳法作为该深度的搜索范围，就不难看出这是一个全排列的搜索问题。然后在搜索马的所有可行路径的过程中加入筛选条件，取出那些满足题意的解，放弃那些不能到达右上角终点的路径，这就达到了本题的解题要求。

在任一位置点上，马可以有 8 种不同的跳出方向，而本题由于限制了马只能向右跳，所以马向左跳的 4 个方向可以不予考虑，即对马在每一位置的下一步可能跳法我们只要在向右跳的 4 个方向中进行搜索就行了。本题采用了常量数组 dx、dy 来表示马跳动一次的坐标位移，其下标 1 到 4 分别对应马向右跳的 4 种方法（以顺时针顺序排列）。

由于题意要求要输出马从左下角起始位置跳达右上角终点位置的每一步坐标变化过程，所以我们不但要设置一个 a 数组用来记录跳法的改变（也是回溯算法的内循环变量值的变化），还需要设置 x、y 两个数组来记录每一步的跳点坐标。这三个数组的下标范围都取 0 到 30 是因为题意规定 m、n 的范围不超过 30，而马向一个方向跳的增量至少为 1，所以马跳到终点不会超过 30 步。x[0]、y[0]用来表示左下角起点坐标（0，0）。

本题的筛选条件是只要保证马的每个跳转位置是在 m×n 的棋盘范围之内即可。程序中设置了 check 函数来完成筛选任务，在 xx 和 yy 的边界条件判断中，因为得到常量数组 dx 的取值始终大于 0 的保证，所以我们省去了对 xx < 0 这一条件是否满足的判断。

**参考程序：**

```
PROGRAM ex1_3_5(input,output);
CONST  {常量数组 dx、dy 依次表示马向右跳的四个可能方向（顺时针序）}
  dx : array[1..4] of shortint = (1,2, 2, 1);
  dy : array[1..4] of shortint = (2,1,-1,-2);
VAR                              {x,y 数组用于记录各次跳点的坐标}
  a,x,y     : array[0..30] of byte;
  xx,yy     : shortint;
  i,j,k,n,m : byte;
  l         : longint;
FUNCTION check : boolean;
  begin  {由于常量数组取值保证了马只能向右跳，所以 xx 不会小于 0}
    if (xx > n) or (yy < 0) or (yy > m) then check := false
                                        else check := true
  end;
PROCEDURE print;
  begin
    l := l+1;
    write('No.',l:2,' : 0,0--');
    for k := 1 to i-1 do write(y[k],',',x[k],'--');
    Writeln(m,',',n);
    if l mod 24 = 0 then readln
  end;
BEGIN
  repeat
    write('M = ');  readln(m);
    write('N = ');  readln(n)
  until (n in [3..30]) and (m in [3..30]);
  x[0] := 0;  y[0] := 0;     {x、y 数组从左下角起点位置开始记录}
  i := 0;  l := 0;                    {l 是统计解的个数的计数器}
  repeat
    i := i+1;  j := 0;
    repeat
      j := j+1;                {内循环在马的 4 种跳法中进行搜索}
      if j <= 4
        then begin
               xx := x[i-1]+dx[j];   {xx、yy 为试解的马的新跳点}
               yy := y[i-1]+dy[j];
```

```
            if check then      {check 函数用于判断马是否跳出界}
              begin    {a 数组记录跳法，x、y 数组记录跳点坐标}
                a[i] := j;  x[i] := xx;  y[i] := yy;
                if (x[i] = n) and (y[i] = m) then print else break
              end      {跳到右上角即输出一组解，否则继续搜索}
          end
        else begin
              i := i-1; {当前层的 4 种跳法搜索完毕即回溯到前一层}
              if i = 0 then halt;
              j := a[i]   {j 从前一层记录的当前跳法位置继续搜索}
          end
    until false
  until false
END.
```

## 例 1-3-6　跳马问题（2）

**问题描述：**

棋盘为 n×n 的正方形，输入 n 值，让马从棋盘的任意位置出发，按“日”字跳走的规则，输出一种能不重复地走遍棋盘上每一点位置的方法（输出方式为：在一个 n×n 的方阵上，左上为坐标（1，1），右下为坐标（n，n），每个位置标示的为步骤值）。为了比较算法的优劣，请再设置一个变量用于统计整个程序中试走的步骤数。

**算法分析：**

本题与上题的主要区别是：

1. 搜索的终止条件不同。上题是跳到一个指定的位置，而本题是要跳满棋盘的所有位置。
2. 要求的解的个数不同。上题要求出所有满足条件的解，而本题只要求出一组满足条件的解。
3. 马的跳行限制不同。上题马只能向右跳，所以只能有 4 个可能的跳马方向，而本题马在 8 个方向上都可以跳。
4. 本题还要求增加一个统计马的试跳步骤的计数器。

本题在程序中专门设置了一个 b 数组用作为控制回溯操作的栈数组。上题的回溯过程终止控制是由变量 i=0 来控制的，而本题却是由栈数组 b 中的栈底元素 b[0]≠0 来控制的。当然由于本题只要求输出一组解，所以在有解的情况下实际终止是由 m=n*n-1（表示马已跳满全盘所有位置）的条件来控制的，只有在无解的情况下栈底元素值的变化才会起控制作用。

本题的输出是采用二维字符图形来表述棋盘上马的各点跳行位置的

顺序的，这样输出既形象又容易理解。

**参考程序：**

```
PROGRAM ex1_3_6(input,output);
CONST                          {常量数组设置了马能够跳行的 8 个方向}
  dx:array[1..8] of shortint=(1,2,2,1,-1,-2,-2,-1);
  dy:array[1..8] of shortint=(-2,-1,1,2,2,1,-1,-2);
  n1=20;
VAR
  m,k,i,n,x,y,x1,y1:integer;
  a                :array[1..n1,1..n1] of integer;{记录跳马步骤}
  b                :array[0..n1*n1] of integer;   {回溯控制数组}
  cr               :longint;
BEGIN
  write('Input n: ');  readln(n);                  {输入棋盘的边长}
  write('Input x,y: ');  readln(x,y);            {输入马的起跳位置}
  m:=0;  k:=0;  cr:=0;   {跳马步骤、跳马方向、试跳总数计数器置 0}
  for i:=0 to n*n do  b[i]:=0;             {回溯控制的栈数组置初值}
  for x1:=1 to n do
    for y1:=1 to n do   a[x1,y1]:=0;   {跳马步骤记录数组置初值 0}
  a[x,y]:=1;                                     {从指定的位置起跳}
  while (b[0]=0) and (m<n*n-1) do   {m=n*n-1 表示已找到了 1 组解}
    begin
      inc(k);                               {马的跳行方向试解计数器}
      if k>8
        then begin、                          {满足条件进行回溯处理}
               a[x,y]:=0;                     {当前跳马步骤值恢复 0}
               k:=b[m];  dec(m);  {前一跳马方向值从回溯数组出栈}
               dec(x,dx[k]);  dec(y,dy[k])    {回到前一跳马位置}
             end
        else begin
               inc(x,dx[k]);  inc(y,dy[k]);        {试解下一位置}
               if (x<1) or (x>n) or (y<1) or (y>n) or (a[x,y]>0)
                 then begin     {如不满足跳行条件则退回原来位置}
                        dec(x,dx[k]);  dec(y,dy[k])
                      end
                 else begin          {满足跳行条件则记录跳马步骤}
                        inc(m);  a[x,y]:=m+1;
```

```
                    inc(cr);              {试跳总数计数器计数}
                    b[m]:=k;  k:=0{当前跳马方向值入栈并重置}
                  end
            end
    end;
  for y:=1 to n do    {用两维字符图形方式输出棋盘跳马步骤示意图}
    begin  for x:=1 to n do  write(a[x,y]:3);  writeln  end;
  writeln('Total step:',cr)                    {输出试解步骤总数}
END.
```

## 例 1-3-7　迷宫问题

**问题描述：**

有一 m×n 的迷宫（3≤m≤24，3≤n≤50)。老鼠在迷宫中可以沿上下左右四个方向摸索前进。编一程序，由计算机自动找出一条从上面入口处到下面出口处的最短路径。

迷宫的数据由文件“mouse.in”输入，第 1 行有两个数据，为迷宫的行数和列数；接下来有 m 行数据，每行有 n 个数据，且所有数据都由 0 和 1 组成，其中 1 表示墙壁，0 表示通路。

对于任意组成的满足题意的 m×n 的迷宫，要求找出一条从上面入口到下面出口的最短通路，并用字符组成图形的方式（“@”表示墙壁，“　”表示通路，“#”表示求出的最短通路）打印出迷宫和最短通路的示意图。

**算法分析：**

本题也是一个典型的全排列搜索问题。与跳马问题不同的是它不要求输出所有的解，而是要求从这些解中筛选出一个搜索深度最浅的解来（即最短路径)。为此我们设置了两组记录路径的数组 x、y 和 xm、ym，前者用于在回溯算法中记录当前搜索到的通路，而后者用于记录当前这些通路中的路径最短者。

由于迷宫的数据用文件方式输入，所以我们设置了二维逻辑型数组 a，其值为 true 表示可行的通路，为 false 表示墙壁不能通行。

题解的输出要求用字符图形的方式表示,而且还要画出一条最短路径，所以我们又设置了二维逻辑型数组 b，其值为 true 表示最短通路。这样，在 a、b 两数组的共同描述下就能画出迷宫和最短路径来。

在本题的每一位置上我们都要考虑从上、下、左、右 4 个方向进行搜索，所以筛选条件主要是判断下个搜索位置是否曾经走过，以避免无限循环的情况出现；当然是通路还是墙壁这一条件是非判断不可的；至于跳马问题中的马是否跳出界外的问题，这里可以不必担心，因为老鼠是一步步前进的，不可能错过下边界 n 而跳出界外。

程序中多处出现 1200 这个数值，它是由迷宫的范围最大值 m×n 得到。坐标数组的下标最大值为 1200 表示深度最大不会超过搜索遍所有的坐标点；至于 min=1200 是为了给最短路径的记录变量设置一个足够大的初始值，它也最大不会超过迷宫所有坐标点的总数。

在算法处理上和跳马问题不同的是：原来只要满足搜索的终止条件就立即打印输出一个解，而这里仅是转去执行子过程 rec，而子过程 rec 的作用是在每一个满足搜索终止条件的可能解中筛选出一个搜索深度最小的（即最短路径）解来；另外一个不同点是，原来当回溯过程执行到 i = 0 时程序运行即终止，而这里还要转去执行 print 子过程，输出以字符图形表示的解以后再终止程序运行。以上两点不同的根本原因是由于本题要求在搜索出来的所有解中还要筛选出一个最优解来。

**参考程序：**

```
PROGRAM ex1_3_7(input,output);
CONST                              {依次按右、下、左、上 的顺序搜索}
  dx : array[1..4] of shortint = (1,0,-1,0);
  dy : array[1..4] of shortint = (0,1,0,-1);
VAR
  a, b            : array[1..24,1..50] of boolean;
  x, y, xm, ym, d : array[0..1200] of byte;
  j,m,min,n,i     : integer;
PROCEDURE init;                                {迷宫数据从文件输入}
  var
    i,j,t : byte;
  begin
    assign(input,'mouse.in');  reset(input);  readln(m,n);
    for i := 1 to m do
      begin
        for j := 1 to n do
          begin
            read(t);                           {0 是通路，1 是墙壁}
            if  t = 0  then a[i,j] := true
                       else  a[i,j] := false
          end;
        readln
      end;
    min := 1200;  close(input)  {min 的初值设为最大的深度可能数}
  end;
```

```
PROCEDURE rec;                    {本过程只记录最短路径于 xm 和 ym 中}
  var
    k : integer;
  begin
    if i < min then      {min 中记录的是所有通路中深度 i 最小的值}
      begin
        min := i;
        for k := 0 to i do{满足条件的 x、y 数组将复制到 xm、ym 中}
          begin  xm[k] := x[k];  ym[k] := y[k]  end
      end
  end;
FUNCTION check : boolean;
  var
    k : integer;
  begin
    check := false; {此条件判断是否可通，不通退出，函数值为 false}
    if not a[y[i-1]+dy[j],x[i-1]+dx[j]] then  exit;
    for k := 1 to i-2 do           {此条件判断该位置是否曾经走过}
      if (x[i-1]+dx[j] = x[k]) and (y[i-1]+dy[j] = y[k]) then exit;
    check := true           {可通行，并且未曾走过，函数值为 true}
  end;
PROCEDURE print;
  begin
    writeln;
    if min = 1200 {满足条件表示未找到任何通路，所以 min 初值未变}
      then writeln('No answer!')
      else begin
             fillchar(b,sizeof(b),false);
             for i := 0 to min do          {记录最短路径于 b 数组}
               b[ym[i],xm[i]] := true;
             for i := 1 to m do   {用字符图形描绘迷宫和最短通路}
               begin
                 for j := 1 to n do
                   if b[i,j]
                     then write('#')   {'#'表示找到的最短路径}
                     else  if a[i,j]
                             then write(' ')       {空格表示通路}
```

```
                              else write('@' );    {'@' 表示墙壁}
                writeln                          {输完一行后要换行}
              end
          end
    halt                                 {输出结果以后结束程序运行}
  end;
BEGIN
  init;                                          {输入 m×n 的迷宫数据}
  y[0] := 1; {在第一行找迷宫入口，记入坐标数组 x、y，坐标为(1,j)}
  for j := 1 to n do  if a[1,j] then x[0] := j;
  i := 0;
  repeat
    i := i+1;  j := 0;
    repeat
      j := j+1;
      if j <= 4              {j 将右、下、左、上 4 个方向搜索一遍}
        then begin    {check 函数判断能否走通，不能：找下一方向；
              if check then                    能：进入该位置。}
                begin
                  x[i] := x[i-1]+dx[j];    {记录进入的位置坐标}
                  y[i] := y[i-1]+dy[j];
                  d[i] := j;
                  if y[i] = m then rec
                              else break
                end       {走通则进入 rec 过程记录通路和距离值 i}
            end
        else begin
              i := i-1;                              {回溯处理}
              j := d[i];
              if i = 0 then print        {搜索完成打印最短路程}
            end
    until false
  until false
END.
```

**附：迷宫文件 mouse.in 的内容如下：**

```
10 21
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1
1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 0 1
1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1
1 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1
1 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 0 1
1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1
1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

以上我们讨论了一些回溯搜索的应用实例，可以看到这些程序中如何用一个数组和两个计数器来有效控制栈形式的数据结构在搜索过程中的处理方法。为了使大家能尽快掌握回溯算法的基本规律，我们在这些例子的编程方法上尽量不作什么变化，也就是说在这些例子的编程处理方法和语句实现方面尽量相类似，这样对大家尽快熟悉回溯算法、掌握编程应用是有好处的。但是我们知道程序设计的变化是丰富多彩的，不可能是形式单一的，如果我们仅仅是背出了某种算法的形式而不是真正掌握了回溯算法的真谛，那是远远不够的。为了能进一步了解回溯算法的多种形式的表现，下面我们再通过一些例子来看看其他形式的编程方法在回溯算法中的应用，以期达到真正掌握回溯算法原理的目的。

## 例 1-3-8　砝码称重问题

**问题描述：**

设有 1 克、2 克、5 克、10 克、20 克、50 克的砝码各若干枚，问用这些砝码可称出多少种不同的重量，设砝码的总重量不超过 1000 克。

**算法分析：**

本题可采用全组合的算法，用数组 d[1] ~ d[6]分别存放每种砝码的个数。

例如：取 d[1]=2，d[2]=3，d[3]=4，d[4]=1，d[5]=0，d[6]=2 为例分析如下：

①开始时：

取 1 克砝码 1 个，可称 1 克重量，此时可称总数为 1；

取 1 克砝码 2 个，可称 2 克重量，此时可称总数为 2；

②此时 1 克砝码已取完，再从 2 克砝码开始取：

取 2 克砝码 1 个，1 克砝码 0 个，可称 2 克重量，由于已存在，此时可称总数仍为 2；

取 2 克砝码 1 个，1 克砝码 1 个，可称 3 克重量，此时可称总数为 3；

取 2 克砝码 1 个，1 克砝码 2 个，可称 4 克重量，此时可称总数为 4；

取 2 克砝码 2 个，1 克砝码 0 个，可称 4 克重量，此时可称总数不增加，仍为 4；

取 2 克砝码 2 个，1 克砝码 1 个，可称 5 克重量，此时可称总数为 5；

取 2 克砝码 2 个，1 克砝码 2 个，可称 6 克重量，此时可称总数为 6；

……

③用此方法直到全部砝码取完，可得到最大称重为：

d[1]+2*d[2]+5*d[3]+10*d[4]+20*d[5]+50*d[6]

=2 + 6 + 20 + 10 + 0 + 100

=138

并统计出可称总数为 116。

上面的算法可通过以下步骤实现：

①引入 b：array[1.. 6] of integer 数组，初始为 0，然后进行逐次加 1 的算法，并设一个变量 j 控制 b 数组的下标，对各种规格的砝码判断是否取完：

```
j:=1;                                  (从 1 克的砝码取起)
while b[j]=d[j] do  inc(j);            (找下一个未取完的砝码)
inc(b[j]);   (逐个取砝码，直至取到与 d[j]相同，表示取完)
for i:=1 to j-1 do  b[i]:=0;                (相当于回溯清零)
```

②取一次砝码即用下面公式计算可称的重量：

```
w:=b[1]+2*b[2]+5*b[3]+10*b[4]+20*b[5]+50*b[6];
```

③对于每次取得的砝码称重是否重复的问题可以引入数组：

```
p：array[1.. 1000] of 0.. 1
```

这样每次计算出的 w 值，仅做 p[w]:=1 即可，避开了判断是否有重复的繁复问题。

④砝码取完，只要将 p 数组求和，就可得到本题的解。

⑤本题的编程中多设置了 b[7]这个数组元素，用于循环搜索的终止条件，当 b[7]=0 时表示取砝码数组 b 还未取完砝码，因为取砝码过程是由 b[1]开始逐级递增，并不断与记录砝码品种的数组 d 进行比较，取完第 1 种再接着取第 2 种，……，直至 b[6]取完，这时若再取就会使 b[7]加 1，改变了 b[7]的值，使循环终止条件满足。

⑥由于我们每取 1 个砝码都要将小于该种砝码的前面品种砝码取用数回溯清零，以使在取用的每个大砝码值上再用前面小的砝码值逐个累加计值，这样才能保证所有能称量的值不会漏掉，只有前面的所有品种的所有砝码都已取完，才会进位取下一个较大品种的砝码。如何保证这种取砝码的过程正确进行呢？程序中我们设置了一重专门的循环

（见程序中括号括起的部分），用 b[j]=d[j]作为循环条件，只要循环条件满足就表示该种砝码已经取完，再判断下一种砝码的取用情况，……，直至顺序找到一个还未取用完的砝码品种，再执行取砝码操作。但是要注意，当 j<7 的时候，执行不会遇到问题，而 j=7 的时候就会出现问题了，一是首先要增加 d 数组的元素个数，设置 d[7]，二是 d[7]的值必须大于 0，能让 b[7]（其值是 0）与其比较时能不等而进位达到循环终止条件的实现。

⑦本题的回溯设计和前面诸题有明显的区别：一是用取砝码数组作为回溯数组，每取一个砝码就将前面所取砝码的数组元素值全部清零，从较小的砝码值开始逐级进位，类似于数值的不同权位逐级进位一样，这正好保证了每一种不同的砝码取值不会遗漏。二是搜索的终止条件设置在回溯数组的高位元素上，不像前面诸题所设计的那样放在低位数组元素上的情况,这也正是题意表示取完砝码就终止搜索所需要的。

**参考程序：**

```
PROGRAM ex1_3_8(input,output);
VAR
  d,b    :array[1..7] of integer;
  p      :array[0..1000] of 0..1; {用于记录能称的重量，能称为 1}
  i,j,w,s:integer;
BEGIN
  write('Input data:');
  for i:=1 to 6 do  read(d[i]);  readln;  {输入各品种砝码的个数}
  d[7]:=1;          {设置 d[7]>0 是为了保证下面括号的循环条件 b[7]
  for i:=1 to 7 do     ≠d[7]，这样可以让 b[7]+1，才能做到进位退出}
    b[i]:=0;  {取砝码记录数组清零。多设置的 b[7]用作回溯结束标志}
  for i:=1 to 1000 do
    p[i]:=0;   {初始全置 0；以后有一个能称的重量则对应元素即置 1}
  while b[7]=0 do    {当 j 进位至 b[7]=1 表示砝码已取完，循环终止}
    begin
      j:=1;              {j 是取砝码数组 b 的指针，这里从 b[1]取起}
      while b[j]=d[j] do ┐ {本循环的作用是去掉已取完的较小的砝
        inc(j);          ┘  码品种,从新的大 1 号的品种再开始取}
      inc(b[j]);{取砝码。每次加 1，直至该品种取完，再取下 1 品种}
      for i:=1 to j-1 do  b[i]:=0; {每取 1 个砝码以前品种回溯清 0}
      w:=b[1]+2*b[2]+5*b[3]+10*b[4]+20*b[5]+50*b[6];{计算砝码值}
      if w>1000 then begin writeln('Weight > 1000 !!!'); halt end
                else p[w]:=1       {记录该砝码值。如有重复也无妨}
```

```
    end;
  s:=0;
  for i:=1 to 1000 do
  inc(s,p[i]);      {将 p[i]为 1 的记录相加即为能称的不同重量总数}
  writeln('Total=',s)                    {输出能称量的不同重量总数}
END.
```

## 例 1-3-9　钱币问题

**问题描述：**

设有 1 分、2 分、5 分、1 角、2 角、5 角、1 元、2 元、5 元、10 元的钱币若干张，设钱币的总数不超过 100 元。问：

(1) 用这些钱币能组成多少种不同面值；

(2) 要求对于给出的某个面值，能求出该面值可以有多少种不同的组合方式产生。

**算法分析：**

本题的算法与上一题的砝码称重问题的算法类似，不同之处仅在要求的第二个问题的处理上。为了能不但求出某种组合币值能否产生，而且还要求出该种组合币值共有多少种不同的组合方法，我们可以将 p 数组改成计数器来解决这个问题。因为在整个回溯搜索的过程中我们实际上已经把各种不同的币值组合方法都搜索了一遍，对某种币值是否产生过我们原来是将对应下标的 p 数组元素置成 1 来记录的，这样只能知道该组合币值是否能产生，却不能知道能产生多少次，如果把该数组元素改成数组计数器的形式，每产生一次就计数一次，就可以不但知道该组合币值是否能产生，而且也能知道共有多少种不同的组合方法了。

**参考程序：**

```
PROGRAM ex1_3_9(input,output);
VAR
  d,b    :array[1..11] of integer;
  e      :array[1..10] of integer;     {用于存放 10 种钱币的面值}
  p      :array[0..10000] of integer;
  i,j,w,s:integer;
BEGIN
  write('Input data:');                        {输入各种钱币的张数}
  for i:=1 to 10 do  read(d[i]);  readln;
  d[11]:=1;        {为保证循环终止条件能正常实现而设置的辅助标志}
  for i:=1 to 11 do  b[i]:=0;  {取钱币数组清零。b[11]为终止标志}
  for i:=1 to 10000 do  p[i]:=0;      {记录能组合币值的数组清零}
```

```
  e[1]:=1;  e[2]:=2;  e[3]:=5;  e[4]:=10;  e[5]:=20;
  e[6]:=50;  e[7]:=100;  e[8]:=200;  e[9]:=500;  e[10]:=1000;
                              {以上 e 数组用于记录 10 种钱币的面值}
  while b[11]=0 do                          {b[11]=1 为循环终止条件}
    begin
      j:=1;                                        {从 1 分的币值取起}
      while b[j]=d[j] do  inc(j); {跳过已经全部取完的较小的币值}
      inc(b[j]);                              {未取完的币值加取 1 张}
      for i:=1 to j-1 do
        b[i]:=0;          {同时将前面已取完的币值数组元素回溯清零}
      w:=0;
      for i:=1 to 10 do
        inc(w,b[i]*e[i]); {累加所有取用的钱币，计算当前组合币值}
      if w>10000     {不超过 100 元则记录该组合币值，否则出错退出}
        then begin  writeln('Money >10000 !!!');  halt  end
        else inc(p[w])  {注意：这里不是仅记录能产生该种组合币值，
    end;                          还记录了能产生几种这样的组合币值}
  s:=0;
  for i:=1 to 10000 do                    {输出能组合的币值种类数}
    if p[i]>0 then inc(s);  writeln('Total=',s);
  write('Input money:');  readln(w);  {输入要求组合种类数的币值}
  writeln('Kind of money ',w,' is ',p[w]){输出纪录的对应数组元素}
END.
```

## 例 1-3-10　背包问题

**问题描述：**

设有 n（1≤n≤100）件物品，每件物品有一个重量和一个价值，分别记为：

$W_1$，$W_2$，……，$W_n$ （重量）　$C_1$，$C_2$，……，$C_n$ （价值）

今有一个背包，其最大载重量为 Xk，要求从 n 件物品中任取若干件（可以一件不取，也可以全取），使得重量之和≤Xk，而价值之和为最大。

输入要求：

第 1 行为物品的件数；

以下 n 行每行有两个数据，第 1 个数据是重量，第 2 个数据是价值。

输出要求：

第 1 行输出背包能装物品的最大价值的值；

第 2 行输出背包能装下的最大价值的若干个物品的编号。

**算法分析：**

设计思想与砝码称重问题等题类似：

①可以用 w，c：array[1.. n] of real 分别表示 n 件物品的重量和价值；

②用 b，d：array[1.. n] of 0.. 1 记录逐个取物品的过程，其中 b 数组记录搜索的变化过程（0 表示未取，1 表示已取），在搜索的过程中同时用 d 数组记录当时的满足条件的最大价值取法，直至搜索结束，d 数组记录的取法即为本题的解。

③本程序的搜索起点与以前诸题不同，不是从 1 开始而是从 n 开始，说明回溯搜索的实质是正确完成全程搜索而并不关心搜索的起点是从哪里开始。当搜索的起点是从 n 开始的话只要注意两个问题即可：一是回溯数组的下标不是递增而是递减；二是回溯清零是将高端下标的数组元素清零而不是将低端下标的数组清零。

**参考程序：**

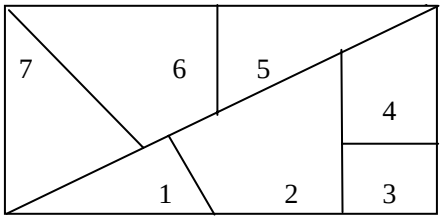
```
PROGRAM ex1_3_10(input,output);
CONST
  m=100
VAR
  w,c          :array[1..m] of real;
  b,d          :array[0..m] of 0..1; {多设置的 b[0]用作终止标志}
  i,j,n        :integer;
  xk,wk,ck,cmax:real;
BEGIN
  write('n=');  readln(n);                    {输入能取的物品数量}
  writeln('No.  W  C');
  for i:=1 to n do                  {输入这 n 种物品的重量和价值}
    begin  write(i:3,':');  readln(w[i],c[i])  end;
  write('Xk=');  readln(xk);                  {输入背包的最大承重}
  for i:=0 to n do    {取物品数组和记录最大价值取法数组置初值 0}
    begin  b[i]:=0;  d[i]:=0  end;
  cmax:=0;                                      {最大价值置初值 0}
  while b[0]=0 do                          {搜索终止条件为 b[0]=1}
    begin
      j:=n;          {与前面诸题不同的是本题从第 n 件物品开始取起}
      while b[j]=1 do  dec(j);                  {查找未取过的物品}
      b[j]:=1;                  {取该物品。0 表示未取，1 表示已取}
      for i:=j+1 to n do  b[i]:=0;  {同时前面已取的物品回溯清零}
      wk:=0;                            {对当前的物品取法计算总重量}
```

```
      for i:=1 to n do  wk:=wk+b[i]*w[i];
      ck:=0;                          {对当前的物品取法计算总价值}
      for i:=1 to n do  ck:=ck+b[i]*c[i];
      if (wk<=xk) and (ck>cmax) then
        begin  cmax:=ck;  d:=b  end   {记录不超重、价值大的取法}
    end;
  writeln('Cmax=',cmax:0:2);                {输出能取的最大价值}
  for i:=1 to n do                     {输出该最大价值的物品取法}
    if d[i]=1 then write(i:3);  writeln
END.
```

## 例 1-3-11 四色问题

**问题描述：**

设有如图所示的具有 n（1≤n≤100）个区域的地图，每个区域的数字代表该区域的编号，现在在每个区域中涂上红（R)、黄（Y)、兰（B)、白（w）四种颜色之一，要求相邻的区域涂的颜色不能相同，请找出一种涂色方案来。




输入输出样例：

用提供的文本文件 map.txt 为数据（表示的是题示的图形）输入，其文件内容如下：

```
7                                {第1行的数据表示要涂色的区域数n}
0 1 0 0 0 1 1                {从第 2 行开始的 n 行为区域的邻接矩阵}
1 0 1 1 1 1 0
0 1 0 1 0 0 0
0 1 1 0 1 0 0
0 1 0 1 0 1 0
1 1 0 0 1 0 1
1 0 0 0 0 1 0
```

要求的输出结果为：(先输出邻接矩阵，再输出各区域的编号和颜色)

```
n = 7
0 1 0 0 0 1 1
1 0 1 1 1 1 0
0 1 0 1 0 0 0
0 1 1 0 1 0 0
0 1 0 1 0 1 0
1 1 0 0 1 0 1
1 0 0 0 0 1 0

Map color：
1：R
2：Y
3：R
4：B
5：R
6：B
7：y
```

**算法分析：**

① 四种颜色可以用数字 1、2、3、4 代表，但输出时应转换为 R、Y、B、W 的文字形式；

② 各区域之间的关系可以用邻接矩阵表示，置 1 表示两区域相邻，置 0 表示不相邻。

③ 由于邻接矩阵数据巧妙地用 0 和 1 两个数字来表示不相邻和相邻，所以对 i 区域能否涂 j 色的判断就可以通过条件 s[k]*r[i,k]是否等于 j 来决定，相等表示涂色冲突，不能涂 j 色。我们可以这样考虑：
当 i 和 k 区域不相邻时，r[i,k]=0，则 s[k]*r[i,k]<>j，可以涂 j 色；
当 i 和 k 区域相邻时，r[i,k]=1，则 s[k]*r[i,k]就等于 s[k]，表示 k 区域所涂颜色值只有为 j 时 s[k]*r[i,k]才能等于 j，正好满足涂色冲突，其他颜色则不会冲突。

④ 检查涂色冲突与否的另一个条件 k<i 是表示对 k 区域的搜索只要在比当前区域号 i 小的区域范围内进行即可，因为这时大于 i 号的区域还未设定色号。

**参考程序：**

```
PROGRAM ex1_3_11(input,output,ft);
CONST
  m=100
```

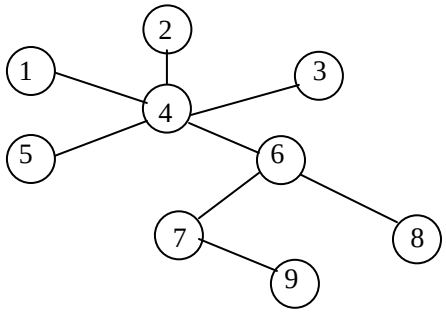
```
VAR
  r    :array[1..m,1..m] of 0..1;                    {邻接矩阵数组}
  n,i,j:integer;
  s    :array[0..m] of integer;                  {区域涂色记录数组}
  ft   :text;
PROCEDURE mapcolor;                                  {回溯搜索过程}
  var
    i,j,k:integer;
  begin
    s[1]:=1;                                {第 1 个区域作第 1 种颜色}
    i:=2;                             {待涂色区域计数器指向第 2 个区域}
    j:=1;                                          {颜色计数器置初值}
    while i<=n do                        {循环对 n 个区域进行涂色处理}
      while (j<=4) and (i<=n) do
        begin
          k:=1;{k 指向已涂色的区域。初值为 1 表示从第 1 个区域查起}
          while (k<i) and (s[k]*r[i,k]<>j) do
            inc(k); {检查与 i 相邻区域的涂色情况，判断能否涂 j 色}
          if k<i then inc(j) {k<i 表示不能涂 j 色，再试下一种颜色}
                 else begin  {k=i 退出表示无涂色冲突，可以涂 j 色}
                        s[i]:=j;                      {i 区域涂 j 色}
                        inc(i);                    {i 指向下一区域}
                        j:=1                      {色号指针重新置 1}
                      end;
          if j>4 then begin  {j>4 表示取色失败，回溯重取前一颜色}
                        dec(i);                       {返回上一区域}
                        j:=s[i]+1              {重新试区下一种颜色}
                      end
        end
  end;
BEGIN
  assign(ft,'map.txt');  reset(ft);
  readln(ft,n);  writeln('n=',n);      {从文件输入待涂色区域数 n}
  for i:=1 to n do
    begin
      for j:=1 to n do                           {输入邻接矩阵数据}
        begin  read(ft,r[i,j]);  write(r[i,j]:2);  end;
```

```
      readln(ft);  writeln
    end;
  close(ft);
  mapcolor;              {进入回溯搜索子过程，决定各区域的涂色值}
  writeln('Map color:');
  for i:=1 to n do                          {输出各区域的涂色结果}
    begin
      write(i,':  ');
      case s[i] of
        1:writeln('R');
        2:writeln('Y');
        3:writeln('B');
        4:writeln('W')
      end
    end
END.
```

## 例 1-3-12　无根树的编码问题

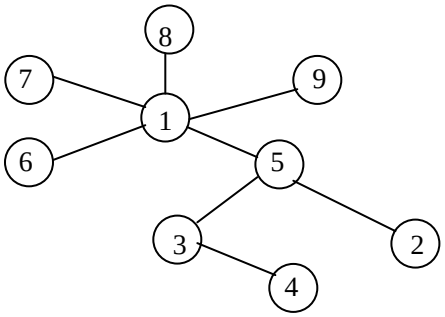
**问题描述：**

设有 n（1≤n≤100）个结点（编号为 1，2，……，n）的无根树如图所示：(图示的 n = 9)




问题一：(文件名为：ex1_3_12a.pas)

现在要求给每个结点重新编号，编号仍为 1，2，……，n。但要求新编的编号能使相邻的结点的编号差的绝对值正好构成一个公差为 1 的等差数列，如下图所示为一种解法，请求出一种满足条件的解：




问题二：(文件名为：ex1_3_12b.pas)

现在要求给每个结点重新编号，编号改为 0，1，2，……，n – 1 。仍要求新编的编号能使相邻的结点的编号差的绝对值正好构成一个公差为 1 的等差数列，并请求出所有满足条件的解和统计解的总数。

**算法分析：**

问题一：

①可以用邻接矩阵 r：array[1.. n，1.. n] of 0.. 1 表示无根树的相邻关系，结点之间有连接用 1 表示，没连接用 0 表示；

②用数组 s：array[1.. n] of integer 存放重排的结点编码；

③编制一个能判断是否构成等差数列的函数如下：

```
FUNCTION f:boolean;
  var {t 数组下标对应相邻两结点的差的绝对值，置 1 表示有此值}
    t    :array[1..n] of integer;
    i,j,w:integer;
  begin
    for i:=1 to n-1 do  t[i]:=0;          {初始 t 数组全置 0}
    for i:=1 to n do                      {对所有结点根据邻接矩阵表示的关
      for j:=1 to n do                    系对 t 数组赋值 :将有相邻关系的结
        if r[i,j]=1 then                  点下标差所对应的 t 数组元素置 1}
          begin  w:=abs(s[i]-s[j]);  t[w]:=1  end;
    w:=0;
    for i:=1 to n-1 do inc(w,t[i]);   {统计 t 数组的赋值结果}
    if w=n-1 then f:=true {若 n-1 个值全存在表示构成等差数列}
             else f:=false
  end;
```

④算法设计思想为：

让 s 数组存放的结点编码值从升序排列开始，逐个从低位向高位进行结

点编码值交换移动，直至达到 s 数组的结点编码值为降序排列为止，这样可保证搜索到各种可能的结点编码排列情况。且每交换移动一次结点编码数据即判断是否满足等差数列的要求，直至找到第一个解。程序如下：

```
    for i:=1 to n do s[i]:=i;                {初始 s 数组升序排列}
    while not f do
      begin
        j:=n;          {找出 s 数组满足降序排列的尾部段前边界点 j}
        while s[j]<s[j-1] do dec(j);  p:=j;
        for i:=j+1 to n do
          if (s[i]>s[j-1]) {and (s[i]<s[p])} then p:=i;
   {此循环的作用是在 s 数组的尾部降序排序段中找出与不满足降序排
    序关系的前部段尾点 j－1 处值进行交换的位置。它的值应比 j－1
    处的值大，但又是降序排列段中编码值尽可能小的}
        t:=s[j-1];  s[j-1]:=s[p];  s[p]:=t;   {交换两点的编码}
        for i:=j to n-1 do          {再对原降序序列进行升序排序。
          for k:=i+1 to n do        它类似于回溯算法中的回溯作用，表
            if s[i]>s[k] then       示重新从最小的排列情况退回重来}
              begin  t:=s[i];  s[i]:=s[k];  s[k]:=t  end
      end;
```

问题二：

①要在问题一的基础上再求出所有的解来，搜索的终止条件改变了，所以我们在找到一组解的时候不能结束搜索过程，只能将此组解打印输出后继续搜索。所以程序中专门设置了一个打印子程序完成此任务。

②搜索的终止条件可以让 s 数组从全升序排列开始搜索到全降序排列结束。为了控制搜索过程的结束，我们设置了函数 fend 对当前的 s 数组状态随时进行判断，再设置了一个全降序排列的标志数组 send，只要 s 数组的状态和 send 标志数组一致，就表示搜索任务已经完成，搜索可以终止。

③当然本题还要注意编号的范围和前题不同，是从 0 到 n-1。

**问题一参考程序：**

```
PROGRAM ex1_3_12a(input,output,ft);
CONST
  m=100
VAR
  r          :array[1..m,1..m] of 0..1;          {邻接矩阵数组}
  s          :array[1..m] of integer;
```

```
  n,i,j,k,p,t:integer;
  ft         :text;
FUNCTION f:boolean; {本函数用于判断当前结点编码是否满足等差数列}
  var
    i,j,w:integer;
    t    :array[1..m-1] of integer;
  begin
    for i:=1 to n-1 do  t[i]:=0;{记录邻接结点编码差值的数组置初值}
    for i:=1 to n do        {记录有邻接关系的结点编码差的绝对值}
      for j:=1 to n do
        if r[i,j]=1 then
          begin  w:=abs(s[i]-s[j]);  t[w]:=1  end;
    w:=0;
    for i:=1 to n-1 do  inc(w,t[i]);          {统计编码差的种数}
    if w=n-1 then f:=true           {种数为 n-1 表示等差数列成立}
             else f:=false
  end;
BEGIN
  assign(ft,'tree.txt');  reset(ft);
  readln(ft,n);  writeln('n=',n);           {从文件输入结点个数}
  for i:=1 to n do                      {输入结点关系的邻接矩阵}
    begin
      for j:=1 to n do
        begin  read(ft,r[i,j]);  write(r[i,j]:2);  end;
      readln(ft);  writeln
    end;
  close(ft);
  for i:=1 to n do s[i]:=i;  {s 数组初始化。从全升序排列开始搜索}
  while not f do            {循环终止条件为找到一组满足条件的解}
    begin
      j:=n;  while s[j]<s[j-1] do dec(j);  p:=j;
      for i:=j+1 to n do    {因为是降序序列，所以条件 2 可以省略}
        if (s[i]>s[j-1]) {and (s[i]<s[p])} then p:=i;
      t:=s[j-1];  s[j-1]:=s[p];  s[p]:=t;
      for i:=j to n-1 do
        for k:=i+1 to n do
          if s[i]>s[k] then
```

```
            begin  t:=s[i];  s[i]:=s[k];  s[k]:=t  end
    end;
  for i:=1 to n do                    {输出找到的一组满足条件的解}
    writeln(i,':  ',s[i]:3);  writeln
END.
```

**问题二参考程序：**

```
PROGRAM ex1_3_12b(input,output,ft);
CONST
  max=100
VAR
  r            :array[0..max-1,0..max-1] of 0..1;
  s,send       :array[0..max-1] of integer;
  m,n,i,j,k,p,t:integer;
  ft           :text;
FUNCTION fend:boolean;
  begin
    fend:=true;
    for i:=0 to n-1 do
      if s[i]<>send[i] then fend:=false
  end;
FUNCTION f:boolean; {本函数用于判断当前结点编码是否满足等差数列}
  var
    u,v,w:integer;
    t    :array[1..m-1] of integer;
  begin
    for u:=1 to n-1 do  t[u]:=0;{记录邻接结点编码差值的数组置初值}
    for u:=0 to n-1 do        {记录有邻接关系的结点编码差的绝对值}
      for v:=0 to n-1 do
        if r[u,v]=1 then begin  w:=abs(s[u]-s[v]);  t[w]:=1  end;
    w:=0;  for u:=1 to n-1 do  inc(w,t[u]);   {统计编码差的种数}
    if w=n-1 then f:=true           {种数为 n-1 表示等差数列成立}
             else f:=false
  end;
PROCEDURE swap(x,y:byte);                    {交换两数组元素的值}
  begin
    t:=s[x];  s[x]:=s[y];  s[y]:=t
  end;
```

{本函数用于判断当前的数组状态是否已达到全降序排列的状态，如达到表示搜索过程已经完成，搜索终止}

```
PROCEDURE print;      {对搜索到的满足条件的解输出当前的数组编号}
  begin
    if m mod 23=0  then
      begin             {换屏后每屏显示的第 1 行输出结点原始编号}
        readln;  write('Number:');
        for i:=0 to n-1 do  write(i:3);  writeln
      end;
    write('No ',m:3,':');   {输出当前的一组满足等差数列条件的解}
    for i:=0 to n-1 do  write(s[i]:3);  writeln
  end;
BEGIN
  assign(ft,'tree.txt');  reset(ft);
  readln(ft,n);  writeln('n=',n);               {文件输入结点数}
  for i:=0 to n-1 do
    begin           {以 0 到 n-1 为 n 个结点进行编号并输入邻接矩阵}
      for j:=0 to n-1 do
        begin  read(ft,r[i,j]);  write(r[i,j]:2)  end;
      readln(ft);  writeln
    end;
  close(ft);  m:=0;                  {统计解的个数计数器置初值零}
  write('Number:');                      {第 1 行输出结点原始编号}
  for i:=0 to n-1 do
    begin
      s[i]:=i;                              {s 数组进行升序初始化}
      send[i]:=n-1-i;       {同时建立控制搜索终止状态的降序数组}
      write(s[i]:3)
    end;
  writeln;
  repeat
    if f then  {随时对 s 数组的结点编号进行判断，是解即统计、输出}
      begin  inc(m);  print  end;
    j:=n-1;  while s[j]<s[j-1] do dec(j);  p:=j;{找 j-1 结点位置}
    for i:=j+1 to n-1 do
      if (s[i]>s[j-1]) and (s[i]<s[p]) then p:=i; {找 p 结点位置}
    swap(j-1,p);                    {交换 j-1 结点和 p 结点的编号值}
    for i:=j to n-2 do{用选择排序法对原降序排列的尾部进行升序重排}
      for k:=i+1 to n-1 do
```

```
        if s[i]>s[k] then swap(i,k)
  until fend;                    {搜索达到 s 数组全降排列时结束}
END.
```

**附：无根树的邻接矩阵文件 tree.txt 的内容：**

```
9
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0
1 1 1 0 1 1 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 1 1 0
0 0 0 0 0 1 0 0 1
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
```

# 第四节　深度优先搜索的数据结构

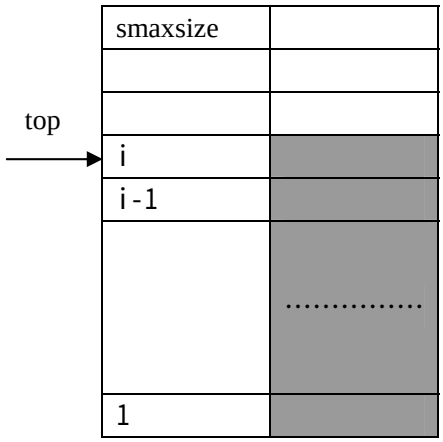
本节我们来讨论一下深度优先搜索的数据结构类型。深度优先搜索是采用堆栈形式的数据结构来存放数据的。在递归搜索的算法中我们利用的是系统设置的堆栈结构，因为每进入一层递归子程序系统会将本层的所有局部变量独立存放，待递归过程回归退出到本子程序时再拿出来使用，这就是一个典型的先进后出的堆栈数据结构形式；而在回溯搜索的算法中，我们直接是用数组和计数指针来构造了一个先进后出的数据表，它也具有堆栈的数据结构所具有的所有特性，是一个人工构造的堆栈。为了对深度优先搜索能理解得更加透彻，我们应该对堆栈的数据结构知识有一个更进一步的全面的了解。

## 一、栈的概念

栈(stack)是一种线性表。是限制只能在表的一端进行插入和删除操作的线性表。

我们把栈中能进行插入和删除的一端称为栈顶(top)，而另一固定端称为栈底(bottom)。

通常堆栈可以采用顺序的方式存储数据。我们可以分配一块连续的存储空间来存放栈中的元素，再用一个指针变量 top 指向当前的栈顶元素。如图所示的灰色背景部分表示已存放了数据，栈的最大容量为 smaxsize，当前栈顶指针指着栈的第 i 个存储空间位置。




栈 s

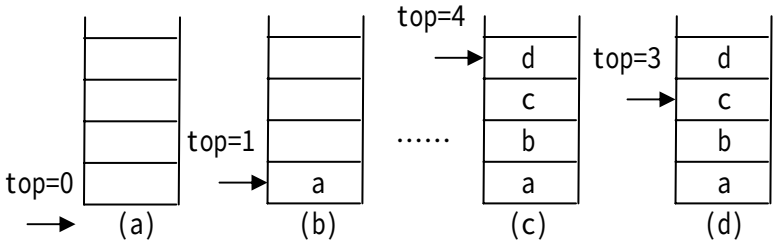
把一个元素放人栈中的操作叫做进栈或压栈(push)，从栈中取出一个元素的操作称为出栈或称弹出(pop)。

直观地讲，栈的进栈操作就象往冲锋枪弹匣中压人子弹(装弹)，栈的

出栈操作就象从弹匣弹出子弹(发射子弹)。也就是说栈中能直接存取的位置是栈的顶部。

根据以上堆栈的定义限制，栈的存取操作必须符合下面的规律：每次出栈的总是当前栈中“最年轻”的栈顶元素，即最后一个进栈的元素；最先进栈的元素则被放在栈的底部，要到最后才能出栈，最后放人栈中的元素则被首先取出，故栈又称为后进先出(LIFO：Last In First Out)线性表(有的书上称为先进后出线性表)。

栈的基本操作如图所示：




(a)空栈。栈指针指向 0；

(b)字符‘a’进栈。栈指针加 1 指向 1；

(c)字符‘b’,‘c’,‘d’依次进栈。栈指针指向 4；

(d)字符‘d’出栈。栈指针退 1 指向 3。

我们要注意的是：当执行(d)步时，栈顶指针 top 倒退了一步，尽管原单元中的内容并未擦去，仍留有‘d’的内容，但对栈这个结构来说，已经不能再对它进行存取。也就是说‘d’已不再存在于栈中了。

在 Pascal 语言中我们可以用一维数组来实现栈的操作。我们先对栈的结构类型和有关变量定义如下：

```
CONST
  smaxsize={栈的最大容量};
TYPE
  selement={元素的数据类型};
  sposition=0..smaxsize;
  stack=record
          data:array[1..smaxsize] of selement;
          top :sposition
        end;
  stackerror=(noerro,empty,stackunderflow,stackoverflow);
VAR
  s    :stack;
```

```
  serro:stackerro；
```

## 二、栈的基本操作

栈的基本操作常常定义成函数和过程，参考程序如下：

1．初始化操作的过程：

```
  PORCEDURE screate(var s:stack)；                    {设定一空栈}
    begin
      s.top:=0；
      serro:=noerro
    end；
```

2．判断栈是否为空的函数：

```
  FUNCTION sempty(var s:stack):boolean； {如果为空函数值为 true}
    begin
      sempty:=(s.top=0)
    end；
```

3．判断栈是否为满的函数：

```
  FUNCTION sfull(var s:stack):boolean；  {如果为满函数值为 true}
    begin
      sfull:=(s.top=smaxsize)
    end；
```

4．元素压栈的过程：

```
  PORCEDURE spush(e:selement；var s:stack)；
    begin  {如果栈已满则指示栈满 stackoverflow,否则把 e 压入栈中}
      if sfull(s)
        then serro:=stackoverflow
        else begin
               s.top:=s.top+1；
               s.data[s.top]:=e
             end
    end；
```

5．元素出栈的过程:

```
  PORCEDURE spop(var e:selement；var s:stack)；
    begin                {如果栈已空则指示栈空 stackunderflow，
      if sempty(s)        否则栈顶元素出栈，值赋给 e}
        then serro:=stackunderflow
        else begin
               e:=s.data[s.top]；
```

```
            s.top:=s.top-1
          end
  end;
```

6. 取栈顶元素值的过程:

```
PORCEDURE stop(var e:selement;var s:stack);
  begin  {如果栈不空，则把值赋给 e，但栈顶元素不出栈}
    if sempty(s)
      then serro:=empty
      else e:=s.data[s.top]
  end;
```

# 第二讲　广度优先搜索

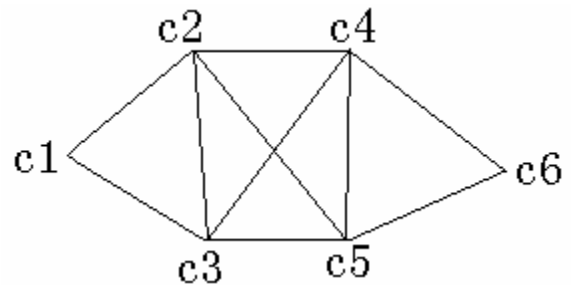
## 第一节　广度优先搜索

在深度优先搜索中，深度越大的结点越先得到扩展；如果将深度越小的结点越先得到扩展，那便是广度优先搜索。广度优先搜索类似于树的按层次遍历的过程，它从初始点开始，在队列中生成第一层结点，检查目标结点是否在这些后继结点中；若没有，再将队列中所有第一层的结点逐一扩展，生成第二层结点，并逐一检查第二层结点中是否包含目标结点；若没有，再将队列中所有第二层的结点逐一扩展，生成第三层结点，并逐一检查第三层结点中是否包含目标结点；……如此依次扩展、检查下去，直到发现目标结点为止。广度优先搜索为了满足先生成的结点先扩展的原则，采用队列的数据结构来存贮结点。队列是一种线性表，对于它的操作具有这样的特点：所有的进队都在线性表的表尾一端进行，所有的出队都在线性表的表首一端进行。如同现实生活中的等车、买票的排队，新来的总是加入队尾，每次离开的总是队首的人。下面我们先通过一个简单的例子来了解一下广度搜索的基本算法形式。

### 例 2-1-1　城市通路问题

**问题描述：**

下面是六个城市之间道路联系的示意图，连线表示两城市之间有道路相通。请编一程序，由计算机找出从 C1 城到 C 6 城的没有经过重复城市的所有不同的路径。




**算法分析：**

要求出所有从 C1 城到 C 6 城的没有经过重复城市的所有不同的路径显然是一个搜索问题，这里我们采用广度搜索的方法来解答本题的问题。

①广度搜索的数据结构与深度搜索不同，它是采用一种称之为队列的线性表来逐层拓展各结点的。我们可以设置一个用于存放队列各结点元素的数组 qm，再在队列中设置两个指针，一个叫队首指针 head,一个叫队尾指针 tail,初始时设置 head=0，tail=1，然后将第 1 个结点 C1 进队。

②为了记录每一个结点是从哪一个结点拓展出来的，我们还要设置一个记录父辈结点地址的数组 pnt，将每一个进队待拓展的结点它原来是从哪一个结点拓展出来的记录在父辈结点地址数组中，以便于搜索到目标城市的解以后能沿父辈结点地址记录的值逐层返回，当然对第 1 个进队的没有父辈结点的结点我们可以设置其 pnt[1]=0,它可以用作倒串返回循环中的循环终止条件。

③至于个城市之间的连接关系我们可以和以前诸题一样设置一个邻接矩阵来表示，矩阵元素值为 1 表示两城市直接连接，矩阵元素值为 0 表示两城市间不直接连接。由于 C6 是终点城市,不需拓展，所以我们可以省略邻接矩阵中 C6 与其他城市的连接情况。注意：其他城市与 C6 的连接情况还是要考虑的。

④为了判断在拓展搜索的路径中是否会产生在已经走过的结点上重复再走得死循环问题，我们设置了存放每个当前结点在拓展之前已经走过的所有结点的集合的数组 f，这样就可以避免出现在一条通路中重复经过同一个城市的问题。

⑤广度搜索的方法是将进队的每一个结点逐个从队首取出进行拓展，只要是和它相连又未曾走过的结点就要安排该结点进队、在 pnt 数组中记录父辈结点地址、在 f 数组中增加一个经过的城市、判断该结点是不是解等操作，如果是解还要打印输出；当一个结点的所有关联城市都拓展完了，我们再从队首取一个结点出队进行下一层拓展；……；直至所有结点都已拓展完毕，head >= tail 时广度搜索终止。

⑥对于解的输出我们设置了一个 print 子过程来处理此任务。由于找到的解是拓展出来的终点城市，我们还必须沿着父辈结点地址数组记录的地址值倒串回去找到首结点，然后再从首结点开始打印输出解的结果来。

**参考程序：**

```
PROGRAM ex2_1_1(input,output);
CONST
  link : array[1..5,1..6] of byte = ((0,1,1,0,0,0),
                                     (1,0,1,1,1,0),
                                     (1,1,0,1,1,0),
                                     (0,1,1,0,1,1),
                                     (0,1,1,1,0,1));
```

{用邻接矩阵表示城市之间的通路情况。其中 c6 因为是终点，所以不必考虑它的邻接边的情况}

```
TYPE
  ft = set of 1..6;                    {设置代表 6 个城市的集合类型}
VAR                          {qm 是队列数组；pnt 是父辈结点地址数组}
  qm,pnt        : array[1..100] of byte;
  f             : array[1..100] of ft;
```

```
  fs            : ft;
  i,k,head,tail : byte;
PROCEDURE print;  {输出子程序的作用是输出一条从起点到终点的路径}
  var
    n,i,j : byte;
    s     : array[1..6] of byte;
  begin
    i := tail;                    {从终点结点开始倒过来查找首结点}
    n := 0;                                  {搜索深度计数器置初值}
    while i > 0 do  {通过父辈结点地址数组倒串记录通路各结点的值}
      begin  n := n+1;  s[n] := i;  i := pnt[i]  end;
    write('Dep = ',n-1,' : 1');    {n-1 是深度，1 是起点城市编号}
    for j := n-1 downto 1 do  {再从首结点开始逐个输出通路各结点}
      write(' -> ',qm[s[j]]);
    readln
  end;
BEGIN
  f[1] := [1];      {记录第 1 个结点城市}
  qm[1] := 1;       {第 1 个结点进队}
  pnt[1] := 0;      {第 1 个结点城市的父辈结点地址设置为 0}
  head := 0;        {队首指针置初值 0}
  tail := 1;        {队尾指针置初值 1}
  repeat
    inc(head);  k := qm[head];                    {将队首元素出队}
    if k <> 6 then                  {如果不是终点城市则进行拓展}
      begin
        fs := f[head];    {此集合赋的是通路中经过的城市结点集合}
        for i := 2 to 6 do      {在所有可能连接的城市中进行搜索}
          if (not(i in fs)) and (link[k,i] > 0) then
            begin     {只要未曾走过又有通路连接的结点就进行拓展}
              inc(tail);  qm[tail] := i;    {满足条件的结点进队}
              f[tail] := fs+[i];{记录当前通路经过的所有结点城市}
              pnt[tail] := head;      {记录该结点的父辈结点地址}
              if i = 6 then print   {找到通到终点的解即打印输出}
            end
      end
  until head >= tail                {广度搜索终止条件是队列为空}
```

```
END.
```

通过上述例题，我们看到在广度优先搜索中，我们是将拓展出来的结点存贮在一个称作 qm 的数组里，qm 数组采用“先进先出”的队列数据结构形式，队列中设了两个指针：head 和 tail，它们分别是队首指针和队尾指针。其中 qm[1..head-1]存贮已拓展的结点（即这些结点的子结点已经拓展，结点本身已经由于队首指针的移动而出队）；qm[head..tail]存贮待拓展的结点（即这些结点的子结点尚待拓展）。当 head >= tail 时则表示队列为空，广度优先搜索过程结束。pnt 为父辈结点地址数组，它记录了每个结点的父辈结点位置，当搜索找到目标后，可沿着父辈结点地址逐层倒串回去，找到首结点，再输出解的路径方案。

在广度优先搜索中，由于搜索是按层次进行结点拓展的，所以搜索到的第一个达到目标结点的解，肯定是层次最少的解，本题的第一个解即是 C1 到 C6 之间的最短路径。所以在一些需要求解最优解的问题中，我们经常会使用广度优先搜索的算法。

下面我们再分析一些广度优先搜索的例子：

## 例 2-1-2　01 字符移动问题

**问题描述：**

有一个左边有四个 1 右边有四个 0，中间有一个空格组成的字符串‘1111 0000’，现规定：

① 1 只能向右运动，0 只能向左运动。

② 空格左右的 1 或 0 可以移动，进入空格。

③ 1 可以跳过一个 0，进入空格；0 也可以跳过一个 1，进入空格。

现要求在符合上述规定的方式下，以最少的移动步骤，将其变为字符串‘0000 1111’。编程打印输出移动的每一步过程。

**算法分析：**

本题是要求出移动步骤最少的最优解问题，所以我们应该采取广度优先搜索的方法来进行求解。

①与上题不同的是本题中间的变化过程需要记录的内容比较多，所以队列中的各结点元素只记录一个数值已远远不够，它要求记录下每一步的 0 和 1 以及空格的位置，所以我们把结点元素的类型设置为记录类型。数据域用 string[9]类型的字符串构成，用于存放 01 字符移动的每一步状态；另外我们还要设置一个指针变量 x 来记录当前数据域中的空格位置，这样比较便于程序中进行字符位置交换的处理；至于父辈结点地址的数组，为了简化程序变量结构形式也可以设置为记录的一个域。这样，我们的记录结构就可以设置为三个域，将程序中需要用到的所有数组结构

都包括进去了。

②对于题中所述的三个移动条件我们可以归纳为：1 和右边的空格交换位置、0 和左边的空格交换位置、1 跳过一个 0 和右边的空格交换位置、0 跳过一个 1 和左边的空格交换位置这样四种操作，并将它们设计成四个子过程 opa___opd 来完成各自的操作任务。

③为了防止拓展出来的新状态是以前曾经出现过的重复状态，我们这里需要设置一个专门的子程序 comp 来完成检查任务。因为上题中每一次产生的新结点是城市编号，且只有 6 种不同的编号，情况比较简单，所以我们只要采用对城市编号集合进行判断的方法就能完成任务。但本题却不行，产生的新状态是以 9 个字符的字符串形式表示的，我们必须逐个地去比较队列中曾经出现过的结点的数据域才行。注意：循环比较应该从队列数组中的 1 号结点开始至 tail 结点为止，而不是从 head 结点至 tail 结点的现存队列，因为 1 号结点至 head - 1 号结点虽已拓展出队，但也不能和新状态重复。

④本题只要求找到一种移动步骤最少的方法，而不是全部解的方法，所以不必完成搜索的全过程。虽然在主程序中循环的终止条件仍然是队列为空，但是由于在输出子程序 print 中设置了 halt 命令，程序会在输出了一组最少移动步骤的解以后，立即强行终止。主程序中循环的终止条件并为起最后终止程序运行的作用。

**参考程序：**

```
PROGRAM ex2_1_2(input,output);
TYPE
  a9 = string[9]; {以 9 个字符的串结构来存放每一步的字符移动情况}
  qt = record {结点用记录结构设置数据、父辈地址、空格位置 3 个域}
         a     : a9;
         x,pnt : byte
       end;
VAR
  qm        : array[byte] of qt;
  temp      : qt;
  head,tail : byte;
PROCEDURE print;
  var
    buf : array[1..30] of byte;
    i,j : byte;
  begin
    i := tail;  j := 0;
```

```
    while i > 0 do      {沿父辈地址数组 pnt 倒串记录到数组 buf 中}
      begin  inc(j);  buf[j] := i;  i := qm[i].pnt  end;
    for k := j downto 1 do          {逐层输出 01 字符移动变化过程}
      begin  write('No.',j-k:2,':',qm[buf[k]].a);  readln  end;
    halt              {根据题意输出第 1 个最优解后即终止程序运行}
  end;
PROCEDURE comp;   {判断新状态是否曾经出现过，未出现过则拓展进队}
  var
    i : byte;
  begin
    for i := 1 to tail do     {搜索新状态在队列中是否已经出现过}
      if qm[i].a = temp.a then exit;    {若新状态已出现过则退出}
    inc(tail);                    {否则新状态结点作拓展进队处理}
    qm[tail].a := temp.a;                       {记录当前位置状态}
    qm[tail].x := temp.x;                          {记录空格位置}
    qm[tail].pnt := head;                       {记录父辈结点地址}
    if temp.a = '0000 1111' then print    {达目标位置则打印输出}
  end;
PROCEDURE opa;    {操作一：1 右移一位进入空格位置，空格左移一位}
  begin
    temp := qm[head];                           {暂存出队结点状态}
    with temp do              {满足操作一条件时则进行操作一处理}
      if (x > 1) and (a[x-1] = '1') then
        begin  a[x-1] := ' ';  a[x] := '1';  x := x-1;  comp  end
  end;
PROCEDURE opb;    {操作二：0 左移一位进入空格位置，空格右移一位}
  begin
    temp := qm[head];
    with temp do              {满足操作二条件时则进行操作二处理}
      if (x < 9) and (a[x+1] = '0') then
        begin  a[x+1] := ' ';  a[x] := '0';  x := x+1;  comp  end
  end;
PROCEDURE opc;          {操作三：1 向右跳过一个 0 与空格交换位置}
  begin
    temp := qm[head];
    with temp do              {满足操作三条件时则进行操作三处理}
      if (x > 2) and (a[x-1] = '0') and (a[x-2] = '1') then
```

```
        begin  a[x-2] := ' ';  a[x] := '1';  x := x-2;  comp  end
  end;
PROCEDURE opd;          {操作四：0 向左跳过一个 1 与空格交换位置}
  begin
    temp := qm[head];
    with temp do              {满足操作四条件时则进行操作四处理}
      if (x < 8) and (a[x+1] = '1') and (a[x+2] = '0') then
        begin  a[x+2] := ' ';  a[x] := '0';  x := x+2;  comp  end
  end;
BEGIN
  with qm[1] do     {第一个原始状态结点进队。x 用于记录空格位置}
    begin  a := '1111 0000';  x := 5;  pnt := 0  end;
  head := 0;  tail := 1;                          {队列指针置初值}
  while head < tail  do   {广度搜索终止条件在本程序中未发生作用}
    begin
      inc(head);
      opa;  opb;  opc;  opd   {opa___opd 分别对应四种允许的操作}
    end
END.
```

## 例 2-1-3　魔板问题

**问题描述：**

在魔方风靡全球后，Rubik 先生发明了它的简化版——魔板，如下图所示：

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |

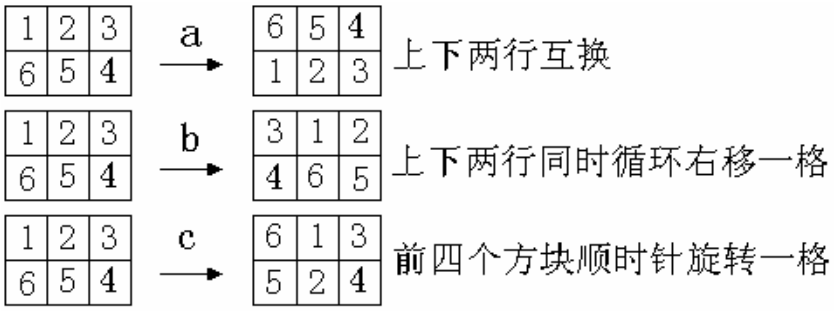
魔板由六个同样大小的方块组成，每个方块的颜色均不相同，本题中用数字 1___6 分别表示这六种颜色，它们在魔板的变化中可能出现在魔板上的任一位置。

对魔板可施加三种不同的操作，分别以 a、b、c 标识，具体操作方法如下图所示：

a 操作：将魔板的上下两行位置互换；

b 操作：将魔板的上下两行各自同时循环右移一格；

c 操作：将魔板的左边（前边）4 个方块顺时针旋转一格；




应用上述三种基本操作，可由任一状态到达任意另一状态。编一程序，对于输入的一个初始状态和一个目标状态，寻找一种最少的操作步骤，打印输出从初始状态到目标状态的每一步操作方法和状态值，并输出最少需要经过多少次操作才能完成任务。

**算法分析：**

本题与上题有很多地方是类似的，下面我们仅对两题不同的地方进行一下分析，其他留给大家自己去分析。

① 本题的三种操作方法比较复杂，能否用数学算法的模型来描述这三种操作方法是解决本题问题的关键。一是可以考虑将六个方块表示成 6 个字符，然后在每一种操作中根据要求对这 6 个字符进行辗转赋值，但这种方法结点的数据域将要设 6 个之多，且上下两行的区别也不方便表明，程序的可读性较差；二是如上题的方法设置两个 string[3]的数据域来存放上下行的字符串，然后在每一种操作中根据要求对这 2 个字符串的元素进行辗转赋值，这样虽然上下两行的区别明显了，但搜索过程中的查找、比较、移动、赋值还是会很繁复；我们这里把上下两行的 6 个字符看成 2 个三位正整数，然后用数学方法根据需要取出要交换的位数值来进行赋值处理，这种方法有几个明显的优点：一是结点的数据域只需要两个整数域即可表述 6 个字符的内容，且在搜索过程中的查找、比较、移动、结点进出队的赋值等都只要对这两个整数进行就可以了，处理简化了很多；二是对三种操作除了 c 操作要各个位数都分解开来进行处理以外，其他两种操作都不需要各个位数全部分解，b 操作的前 2 位不需分解，a 操作则根本不要分解，操作过程可以大为简化；在存储空间的使用上本题的方法也是最节省空间的，因为 1 个整数的占空小于 3 个字符的占空。

② 在数据域中我们设置了一个存放操作方法的域，因为题意要求我们输出每一步使用的操作名称，所以在搜索的过程中我们必须对每一步拓展所使用的操作进行记录，才能完成题意的要求。

③ 本题要求输出最少的步骤数。这实际上就是第 1 个达到目标解的广度优先搜索的深度。

**参考程序：**

```
PROGRAM ex2_1_3(input,output);
TYPE
  qt = record
         x,y : integer;              {用于记录上下两行形成的整数}
         op  : char;                          {用于记录操作方法}
         pnt : integer                      {用于记录父辈结点位置}
       end;
VAR
  qm                : array[1..720] of qt;
  head,tail,k,l,m,n : integer;
PROCEDURE print;
  var
    s   : array[1..30] of integer;
    i,j : integer;
  begin
    j := 0;  i := tail;
    while i > 0 do                  {将解的步骤倒串记录入 s 数组}
      begin  j := j+1;  s[j] := i;  i := qm[i].pnt  end;
    writeln(qm[1].x);                       {首先打印一下初始状态}
    writeln(qm[1].y);  writeln;
    for i := j-1 downto 1 do    {然后逐个打印输出中间的变化过程}
      begin
        writeln('Oprate : ',qm[s[i]].op); {输出使用了哪一种操作}
        writeln(qm[s[i]].x);        {输出经该种操作后的变化结果}
        writeln(qm[s[i]].y);
        readln                      {输出一步操作情况后暂停一下}
      end;
    writeln('Min steps is ',j-1);         {输出最少需经过的步骤数}
    halt  {因只需求出操作步骤最少的解，所以得到解即终止程序运行}
  end;
PROCEDURE comp(opx : char);
  var
    i : integer;
  begin
    for i := 1 to tail do       {循环查找，如出现重复状态则退出}
      if (m = qm[i].x) and (n = qm[i].y) then exit;
```

```
    tail := tail+1;                              {否则将新状态进队}
    qm[tail].x := m;  qm[tail].y := n;       {进队结点数据域赋值}
    qm[tail].op := opx;                 {进队结点记录本次操作方法}
    qm[tail].pnt := head;               {进队结点记录父辈结点地址}
    if (m = k) and (n = l) then print   {找到目标状态则打印输出}
  end;
PROCEDURE opa;                          {a 操作：上下两行位置互换}
  begin
    m := qm[head].y;                        {m 是新产生的上行整数}
    n := qm[head].x;                        {n 是新产生的下行整数}
    comp('a')                    {对 a 操作的结果作能否进队的判断}
  end;
PROCEDURE opb;                  {b 操作：两行各自同时循环右移一格}
  begin
    m := (qm[head].x mod 10)*100+(qm[head].x div 10);
    n := (qm[head].y mod 10)*100+(qm[head].y div 10);
    comp('b')                    {对 b 操作的结果作能否进队的判断}
  end;
PROCEDURE opc;         {c 操作：左边四个位置沿顺时针方向旋转一格}
  var
    a,b,c,d : integer;
  begin
    a := qm[head].x div 100;              {取原上行整数的第 1 位}
    b := qm[head].x mod 100 div 10;       {取原上行整数的第 2 位}
    c := qm[head].y div 100;              {取原下行整数的第 1 位}
    d := qm[head].y mod 100 div 10;       {取原下行整数的第 2 位}
    m := c*100+a*10+(qm[head].x mod 10);        {组成新上行整数}
    n := d*100+b*10+(qm[head].y mod 10);        {组成新下行整数}
    comp('c')                    {对 c 操作的结果作能否进队的判断}
  end;
BEGIN
  write('Input Source M(1,2,3) : ');  readln(qm[1].x);
  write('Input Source N(6,5,4) : ');  readln(qm[1].y);
  writeln;  {以上下两行分别组成两个整数的形式输入初始状态并进队}
  write('Input Object K(1,2,3) : ');  readln(k);
  write('Input Object L(6,5,4) : ');  readln(l);
  writeln;         {以上下两行分别组成两个整数的形式输入目标状态}
```

```
  qm[1].pnt := 0;                          {置初始状态的父辈结点值 0}
  head := 0;  tail := 1;                          {队列指针初始化}
  while head < tail do                          {进行广度优先搜索}
    begin
      head := head+1;          {队首指针加 1 指向待出队拓展的结点}
      opa;  opb;  opc                    {进行三种操作的拓展搜索}
    end
END.
```

## 例 2-1-4　八数码问题

**问题描述：**

八数码问题：在 3 * 3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一个数字，棋盘上留有一个空格，空格周围的棋子可以移到空格中。要求对给出的一种初始状态和一种目标状态，找出一种最少步骤的移动方法，实现从初始状态到目标状态的转变。

例如：

| 初始状态 | 目标状态 |
|---|---|
| 2 8 3 | 1 2 3 |
| 1 6 4 | 8 4 |
| 7 5 | 7 6 5 |

我们至少需经过 5 步移动才能从初始状态达到目标状态。编写程序输出找到的最少移动步骤值和每一步移动的八数码状态变化过程。如未能完成搜索解的任务则输出“No solution !”表示。

**算法分析：**

本题是一个典型的较难的问题。对于任意给定的两个状态并非都是存在解的，所以我们在题中用常量数组的形式提供了一组有解的初始状态和目标状态，供大家调试程序时使用。对于大家输入的其它数据可能会出现以下三种情况：一是有解且解出了最少步骤值；二是无解也说明了结果（但你无法证明无解）；三是有解但由于在有限的内存中无法对结点进行无限制的拓展而解不出解，其结果和第二种情况形式上是一样的。下面我们对本题的算法进行一下分析：

①由于广度优先搜索是分层来进行拓展搜索的，当搜索的层次较深时每一层的结点数量的增加是很可观的。本题的空格移动有 4 个方向可以拓展，也就是说 1 个结点出队，一般情况下将要拓展出 4 个结点进队，而每深入一层就要在上一层的基础上增加 4 倍的一层结点数，其队列结点的总数为：$4^0+4^1+4^2+\cdots\cdots+4^n$ ，当 n 较大时这个数字是很大的，由于受到内存的影响，我们只能解出最少移动步骤在 11 步以内的问题，超

过 12 步就可能会内存不够用了。当然我们直接感觉到受限制的是队列结点最大值的设置，我们在程序中设置为 4800 个结点，如果设置为 5000 个结点就会内存溢出了。

②为了存储每一步的棋盘状态我们设置了一个 3 行 3 列的数组类型来作为各结点的数据域，其中 1 至 8 编码的棋子用 1 至 8 的数字表示，空格用 0 表示，这样数组的数据类型就可以统一设置为 byte 了。我们还利用这一数组类型设置了 start 和 goal 两个常量，前者用于存放初始状态，后者用于存放目标状态。

③本题抓住的解题变化要点是始终关注空格位置的移动，而不是与空格相邻的棋子的移动。程序中与以前诸题一样采用常量数组 dx、dy 来表示空格可能移动的 4 个方向。

④在每个结点中我们还设置了空格位置的指针 x、y，他们分别对应棋盘数组的列号和行好。还有一个域就是父辈结点地址域 pnt。

⑤程序中我们设置了 3 个函数和 2 个过程。check 函数用于判断当前空格准备移动的方向是否可行，也就是会不会移出界；dupe 函数的作用是检查当前进队的结点八数码状态是否和以前队列中曾经记录过的状态相同，以避免出现重复结点拓展进队而造成程序运行的死循环；goals 函数的作用是判断当前进队的结点状态是否和目标结点状态一致，如一致则表示已找到了问题的解；print 过程是用于对找到的目标结点进行按父辈结点地址的倒串搜索方法记录最少移动步骤的每一步过程，再按从初始状态开始一步步变化到目标状态的过程进行输出；最后一个 trace 过程可以跟踪显示广度优先搜索过程中各结点是如何一个个进队、出队的：每个结点是从哪一个结点拓展出来的，是按什么方向拓展的，在队列中是出于什么位置等信息一览无余。不过为了节省程序运行的时间，在主程序中，我们将 trace 过程的执行语句用花括号括了起来，这样 trace 过程就不会被执行了。如果你对广度优先搜索的算法原理和执行过程还不熟悉，想了解比较具体的一步步搜索执行的内容，你可以解除 trace 的花括号，让 trace 过程运行。

⑥至于本题的广度搜索方法与前面诸题原理是一样的，大家可以对程序仔细地进行分析和理解，应该不会有太大的难度。

**参考程序：**

```
PROGRAM ex2_1_4(input,output);
TYPE
  a33  = array[1..3,1..3] of byte;          {棋盘状态的数组类型}
  a4   = array[1..4] of -1..1;      {四个移动方向的常量数组类型}
  node = record                               {队列结点为记录形式}
           ch  : a33;                        {存放当时的八数码状态}
```

```
          y,x : byte;                          {存放当时的空格位置}
          pnt : word                                {存放父辈结点}
        end;
CONST               {为了能形式一致地用数字化表示，空格用 0 代替}
  start : a33 = ((2,8,3),(1,6,4),(7,0,5));            {初始状态}
  goal  : a33 = ((1,2,3),(8,0,4),(7,6,5));            {目标状态}
  dx    : a4 = (0,-1,0,1);    {搜索顺序为上、左、下、右依次进行}
  dy    : a4 = (-1,0,1,0);
  x1 = 2;  y1 = 3;                          {记录空格的初始位置}
  max = 4800;               {为防内存溢出而设置的队列最大允许值}
VAR
  data      : array[1..max] of node;                  {队列数组}
  temp      : node;                      {记录结构的结点暂存变量}
  r,tx,ty   : byte;  {r 是移动方向控制计数器。tx,ty 为新空格位置}
  head,tail : word;
FUNCTION check(k : byte) : boolean;   {判断能否进行 k 方向的移动}
  begin
    ty := temp.y+dy[k];  tx := temp.x+dx[k];      {新的空格位置}
    if (ty in [1..3]) and (tx in [1..3])  {判断空格是否还在界内}
      then check := true      {如新空格位置未出界则允许拓展进队}
      else check := false         {否则不能拓展，该方向移动无效}
  end;
FUNCTION dupe : boolean;      {本函数的作用是检查当前进队结点的八数
  var                         码状态是否与队列中以前结点的状态相同，
    i   : word;               若相同则函数值为 true，否则为 false}
    j,k : integer;
    b   : boolean;
  begin
    i := 0;
    repeat
      inc(i);                   {从队列的第 1 个结点开始进行比较}
      b := true;
      for j := 1 to 3 do          {循环检查棋盘上 9 个位置的数据}
        for k := 1 to 3 do
          if data[i].ch[j,k] <> data[tail].ch[j,k] then b := false
    until b or (i >= tail-1);   {一直查到原队列中的末尾结点为止}
    dupe := b          {如有重复结点则函数值为 true,否则为 false}
```

```
  end;
FUNCTION goals : boolean;        {本函数的作用是检查当前进队结点的八
  var                             数码状态是否与目标状态相同,若相同则
    i,j : byte;                   函数值为 true，否则为 false}
  begin
    goals := false;
    for i := 1 to 3 do {比较当前结点状态与目标状态的 9 个位置数据}
      for j := 1 to 3 do
        if data[tail].ch[i,j] <> goal[i,j]
          then exit;{任一位置数据不同则函数值为 false 退回主程序}
    goals := true  {与目标状态完全一致则函数值为 true 退回主程序}
  end;
PROCEDURE trace;  {此过程可跟踪输出队列中每一步的出队、进队内容}
  var
    i,j : byte;
  begin              {输出当前结点出队、进队和作了那种拓展的信息}
    writeln('head = ',head,'  tail = ',tail,'  oprater = ',r);
    for i := 1 to 3 do  {以二维图形形式输出当前结点的八数码状态}
      begin
        for j := 1 to 3 do
          if data[tail].ch[i,j]=0
            then write(' ')
            else write(data[tail].ch[i,j]);
        writeln
      end;
    readln
  end;
PROCEDURE print;
  var
    b   : array[1..20] of word;
    i,j : word;
    k,n : byte;
  begin
    n := 0;  i := tail;
    while i > 0 do                {对 b 数组倒串赋值记录解的路径}
      begin  inc(n);  b[n] := i;  i := data[i].pnt  end;
    writeln('Staps : ',n-1:5);              {输出最少移动步骤值}
```

```
    for k := n downto 1 do
      begin
        for i := 1 to 3 do  {以二维字符图形方式输出每一步的状态}
          begin
            for j := 1 to 3 do
              if data[b[k]].ch[i,j] = 0
                then write(' ')              {将 0 转换成空格输出}
                else write(data[b[k]].ch[i,j]);
            writeln                          {输完一行数据后换行}
          end;
        readln                           {输完一组状态后暂停一下}
      end;
    halt                 {输出一组最少移动步骤解以后终止程序运行}
  end;
BEGIN
  head := 0;  tail := 1;
  with data[1] do              {进队第一个结点为八数码的初始状态}
    begin  ch := start;  y := y1;  x := x1;  pnt := 0  end;
  repeat
    inc(head);  temp := data[head];                {队首结点出队}
    for r := 1 to 4 do                           {分别搜索四个方向}
      if check(r) then  {check 函数用于判断该方向的移动能否进行}
        begin
          inc(tail);
          data[tail] := temp;{能则作拓展进队处理：这里先原样进队}
          with data[tail] do             {再对进队结点作状态修改}
            begin
              ch[y,x]:=ch[ty,tx];   {原来的空格位置赋新位置的值}
              ch[ty,tx]:=0;                {新位置则赋空格标记 0}
              y := ty;  x := tx;             {记录当前的空格位置}
              pnt := head                      {记录父辈结点地址}
            end;
          {trace;}    {执行此过程可跟踪当前结点的出队、进队内容}
          if dupe  {dupe 函数用于排除与以前相同的八数码状态进队}
            then dec(tail)    {如是重复出现的状态则刚才进队无效}
            else if goals then print  {搜索到目标状态则打印输出}
        end;
```

```
  until (head >= tail) or (tail > max-3);
        {搜索终止条件为队空或队满(不足 4 个空位无法拓展新的结点)}
  writeln('No solution !');        {从终止条件退出表示搜索不成功}
  readln
END.
```

## 例 2-1-5　分酒问题

**问题描述：**

有三个分别能装 8 斤、5 斤、3 斤的酒瓶，其中 8 斤的酒瓶装满了酒，另外两个酒瓶是空瓶，没有其他量器，现要求将 8 斤酒以每瓶 4 斤的量分装在 8 斤和 5 斤的酒瓶内。请设计一程序求出分酒的最少步骤，如存在最少步骤相同的不同解，请分别输出。

**算法分析：**

我们先将解决分酒问题过程中所有倒酒的可能性都列出来，用 c 数组表示 6 种操作序号，用 d 数组表示对应的操作方法，如下表所示：

| 操作数 c[i] | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 操　作 d[i] | 8->5 | 8->3 | 5->8 | 5->3 | 3->8 | 3->5 |

在倒酒的过程中，看起来是每一次倒酒，上面所列的六种操作都有可能进行，然而对应当时的各个酒瓶的盛酒情况，其中的某些操作是不可能进行的。如当 8 斤瓶空时，则进行 1、2 两项操作将该瓶向其它两瓶倒酒就毫无意义；又如当 8 斤瓶满时，若进行将其它瓶的酒倒向该瓶的第 3 项和第 5 项操作也是毫无意义的。所以在每进行一次倒酒操作后，都必须知道此时三个酒瓶里到底有多少酒,这样才能准确判断此时何种操作不能进行，而哪些操作又可以进行。

为了表示每操作一次后各酒瓶里的酒量数，设变量 m 表示 8 斤瓶在进行第 i 次操作后装的酒量，n 表示第 i 次操作后 5 斤瓶装的酒量。由于 3 斤瓶里的酒量是 8 斤瓶和 5 斤瓶里的酒量之和与 8 斤的差数，故此用 a=8–m-n 来表示进行第 i 次操作后 3 斤瓶里的酒量。

这样以上六种操作各自在什么样的条件下可以操作，在什么样的条件下不能操作，我们可设计判断条件如下：

1.进行 8->5 的倒酒操作：

不能进行的条件为：n=5 或 m=0。

否则能进行，则各瓶剩余的数量为：n=n+m，如果 n>=5，则 m=n-5 且 n=5，否则 m=0；

2.进行 8->3 的倒酒操作：
不能进行的条件为：m=0 或 a=3。
否则能进行，则各瓶剩余的数量为：如果 m<=3-a，则 m=0；否则，m=5-n。
注：此时 3 斤瓶必满，5 斤瓶酒量不变。
3.进行 5->8 的倒酒操作：
不能进行的条件为：n=0 或 m=8。
否则能进行，则各瓶剩余的数量为：m=n+m 且 n=0。
4.进行 5->3 的倒酒操作：
不能进行的条件为：n=0 或 a=3。
否则能进行，则各瓶剩余的数量为：如果 n<=3-a，则 n=0；否则 n=5-m，
因为当 n>3-a 时，倒酒以后 3 斤瓶必满，另两个瓶共装 5 斤酒。
5.进行 3->8 的倒酒操作：
不能进行的条件为：a=0 或 m=8。
否则能进行，则各瓶剩余的数量为：m=8-n,因为此时 3 斤瓶必空。
6.进行 3->5 的倒酒操作：
不能进行的条件为：a=0 或 n=5。
否则能进行，则各瓶剩余的数量为：如果 a>=5-n 时 n=5；否则 n=8-m，
因为当 a<5-n 时，倒酒以后 3 斤瓶必空。

对求最少分酒步骤的问题我们可以采用广度优先搜索的算法进行编程设计，找出将 8 斤酒对分为两个 4 斤的最少步骤的分法。我们先设立如下数组变量：

常量字符串 d 数组表示倒酒的六种操作名称；

q 数组的第 2 下标 2 个单元分别表示在倒酒的过程中，每进行一次操作后，8 斤和 5 斤瓶中的盛酒量；

c 数组表示在倒酒的过程中，每进行一次操作的操作数；

b 数组表示每一拓展层的各结点的父辈结点的链接指针；

g 数组表示各层结点的首结点在队列中的位置；

另外为了找出多组可能的解，我们还要设置记录解结果的以下几个数组：

e 数组表示在最少步骤的所有解中，该层各结点是由上一层各结点通过何种操作而得到的操作数；

x 和 y 数组分别表示在最少步骤的所有解中，各层的结点其 8 斤和 5 斤瓶中的盛酒量；a 表示 3 斤瓶的盛酒量；

用 fend 作为整个倒酒操作的结束标记（fend=0 表示搜索结束）；

用 fg 表示在当前的状态下 6 种倒酒操作是否可以进行（1 表示可以进行，0 表示不可以进行）；

另设变量 head 为队列首指针，tail 为队列尾指针，s 为最少步骤的解的方案数，w 为每一层结点的个数，h 为搜索的层数，m 表示 8 斤瓶的盛酒量，n 表示 5 斤瓶的盛酒量。

本题采用广度优先搜索的搜索过程描述如下：

①设立队首队尾指针的初值，即 head=0,tail=1；并使初始状态的结点进入队列，即 q[tail,1]=8,q[tail,2]=0；同时设立其它初值，即 h=1，g[h]=tail+1，w=1。

②取队列首结点，对该状态分别进行 6 种操作，并用对 6 种操作能否进行的条件进行判断。若能进行此操作，则计算进行这种操作后 8 斤，5 斤和 3 斤瓶各自装的数量，同时将此时的各瓶盛酒量和上一层以前的各结点进行比较，若相同则不产生该结点，换另一操作再进行如刚才相同的判断；……；若不同，则将此时产生的结点进入队列，然后继续取另一种操作再重复刚才的过程。

③当 6 种操作能否进行的判断完成之后，则对所产生的该层的所有结点进行判断，是否已出现目标结点。若出现目标结点，则将此时的一条最少步骤的解的各结点按父辈结点链接指针进行链接，并将链接的各结点存储于数组 x[s，i]和 y[s，i]中，同时将到达该结点所进行的操作的操作数存储于 e[s，i]中，同时累计出现目标结点的个数 s，并判断 s 的值是否为 0，若 s=0 则表明没有出现目标结点，转②继续执行；否则转④。

④输出 s 种最少步骤的最优解方案。

**参考程序：**

```
PROGRAM ex2_1_5(input,output);
CONST
  d:array[1..6] of string[4]  {常量字符串数组表示 6 种操作的名称}
    =('8->5','8->3','5->8','5->3','3->8','3->5');
VAR
  q                          :array[1..50,1..2] of shortint;
  b,c                        :array[1..50] of shortint;
  g                          :array[1..20] of shortint;
  e,x,y                      :array[1..10,1..20] of shortint;
  head,tail,h,w,t,i,m,n,a,s,k:integer;
  fg,fend                    :0..1;
PROCEDURE answer(k,h:byte);   {本过程可根据目标结点找出解的路径}
  var
```

```
    i,p:byte;
  begin
    p:=k;
    for i:=h downto 1 do            {按父辈结点指针进行倒串搜索}
      begin
        x[s,i]:=q[p,1];                    {记录 8 斤瓶的盛酒量}
        y[s,i]:=q[p,2];                    {记录 5 斤瓶的盛酒量}
        e[s,i]:=c[p];                      {记录倒酒的操作方法}
        p:=b[p]                                {找父辈结点地址}
      end
  end;
BEGIN
  head:=0;  tail:=1;                          {对列首尾指针置初值}
  q[tail,1]:=8;  q[tail,2]:=0;                  {初始状态结点进队}
  h:=1;  w:=1;  fend:=1;   {h 是层计数器，w 是每层结点个数计数器}
  while fend=1 do                          {fend=0 是搜索终止条件}
    begin
      h:=h+1;  g[h]:=tail+1;       {g 数组记录当前层的首结点位置}
      for i:= 1 to w do                {逐个拓展当前层的各个结点}
        begin
          inc(head);                  {移动队首指针，准备结点出队}
          for t:=1 to 6 do               {逐个进行 6 种操作的处理}
            begin
              m:=q[head,1];  n:=q[head,2];       {取队首结点数据}
              case t of                {对 6 种倒酒操作进行分别处理}
                1:if(m=0) or (n=5)  {1 是 8 斤瓶倒向 5 斤瓶的操作}
                    then fg:=0          {fg=0 表示进行本倒酒操作}
                    else begin
                           n:=n+m;  fg:=1; {fg=1 为可进行本操作}
                           if n>=5 then begin  m:=n-5;  n:=5  end
                                   else m:=0
                         end;
                2:begin                {2 是 8 斤瓶倒向 3 斤瓶的操作}
                    a:=8-m-n;                {求出 3 斤瓶中的盛酒量}
                    if (m=0) or (a=3)
                      then fg:=0
                      else begin
```

```
              fg:=1;
              if m<=(3-a) then m:=0 else m:=5-n
            end
    end;
  3:if (m=8) or (n=0) {3 是 5 斤瓶倒向 8 斤瓶的操作}
      then fg:=0
      else begin  m:=m+n;  n:=0;  fg:=1  end;
  4:begin             {4 是 5 斤瓶倒向 3 斤瓶的操作}
      a:=8-m-n;
      if (n=0) or (a=3)
        then fg:=0
        else begin
               fg:=1;
               if n<=(3-a) then n:=0 else n:=5-m
             end
    end;
  5:begin              {5 是 3 斤瓶倒向 8 斤瓶的操作}
      a:=8-m-n;
      if (m=8) or (a=0)
        then fg:=0
        else begin  m:=8-n;  fg:=1  end
    end;
  6:begin              {6 是 3 斤瓶倒向 5 斤瓶的操作}
      a:=8-m-n;
      if (n=5) or (a=0)
        then fg:=0
        else begin
               fg:=1;
               if a>=(5-n) then n:=5 else n:=8-m
             end
    end
end;
for k:=1 to g[h]-1 do       {检查新结点是否出现过}
  if (q[k,1]= m) and (q[k,2]=n) then
    begin  fg:=0;  k:=g[h]-1  end;{出现则放弃进队}
if fg=1 then              {未出现过则将新结点进队}
  begin
```

```
                tail:=tail+1;                          {移动尾指针}
                q[tail,1]:=m;  q[tail,2]:=n;  {记录结点盛酒量}
                b[tail]:=head;                {记录父辈结点地址}
                c[tail]:=t           {记录当前进行的倒酒操作数}
              end
      end                                    {6 种倒酒操作处理结束}
  end;                                    {当前层的 w 个结点拓展结束}
 s:=0;  w:=tail+1-g[h];                   {计算 h 层生成的结点总数}
 for k:=g[h] to tail do          {在 h 层生成的结点中查找目标结点}
  if (q[k,1]=4) and (q[k,2]=4) then          {若找到则统计解的个
    begin  s:=s+1;  answer(k,h)  end;        数，并记录解的路径}
 if s<>0 then           {解计数器大于零表示有解，则输出解的结果}
   begin
     writeln('best answer:');
     writeln('step:':5,'operate:':10,'8P:':6,'5P:':6,'3P:':6);
     for t:=1 to s do                                {输出 s 组解}
       begin
         for k:=1 to h do  {输出一种倒酒方案。最少步骤数为 h-1}
           begin
             a:=8-x[t,k]-y[t,k];             {计算 3 斤瓶的盛酒量}
             if k=1                         {输出倒酒前的初始状态}
               then writeln(k-1:4,':',x[t,k]:15,y[t,k]:6,a:6)
               else                      {输出每一步的倒酒变化过程}
          writeln(k-1:4,':',d[e[t,k]]:8,x[t,k]:7,y[t,k]:6,a:6)
           end
       end;                                    {s 种解的方案输出结束}
     fend:=0                                             {置终止标记}
   end                        {s 若为零则本条件不执行，继续搜索解}
 end                                                     {搜索结束}
END.
```

通过以上例题我们已经可以基本上弄清了广度优先搜索的基本方法，其中主要的关键是要掌握结点的拓展规律。哪些结点需要拓展，哪些结点不能拓展一般要根据题意要求。一般来讲上述几个例题的结点拓展要求还是比较明确和容易理解的，我们在设计算法时对问题的解决也不感到困难，但有些问题就具有一定的难度了，我们再看下面的一个例题：

## 例 2-1-6　最少转弯问题

**问题描述：**

在一个 m×n（m,n<=20）的方格平面中，有若干个方格是障碍物（用 0 表示），其余方格是通路（用 1 表示）。现给出任意两点 p 和 q ，要求找出一条从起点 p 到终点 q 的转弯最少的路径来。

方格平面数据域由文件 data.txt 输入：第 1 行为 m 行和 n 列的具体数值；以下为 m 行数据，每行有 n 个 0 和 1 表示通路或障碍物；接下来一行是起点坐标的行列值；最后一行是终点坐标的行列值。

要求输出最少转弯数是多少；

要求输出从起点到终点的转弯变化路径。

输入文件样例：

```
6 9
1 1 1 1 1 0 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 0 1 1 1 1
1 1 0 1 1 1 1 1 1
1 1 1 1 1 0 1 0 1
1 1 1 1 1 1 1 1 1
3 2
5 7
```

输出结果：

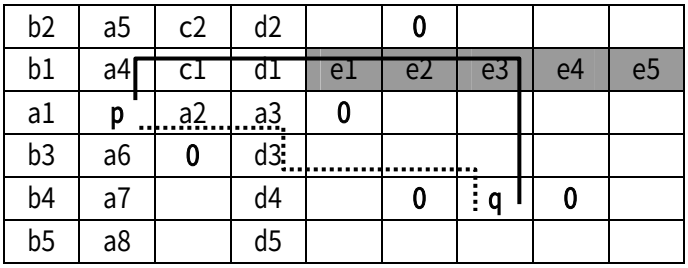
```
Total Across=2
(3, 2) -->(2, 2) --> (2, 7) --> (5, 7)
```

或：

```
Total Across=2
(3, 2) -->(6, 2) --> (6, 7) --> (5, 7)
```

**算法分析：**

本题的结点拓展规律有点特殊，如果不是求最少转弯路径而是求最短通路的话则本题的要求和前面诸题就没有什么大的区别了。为什么呢？我们先看下面的图示：



图中的 p 点和 q 点分别表示起点和终点，图中的障碍物都用 0 表示。显然图中的虚线表示 p 点到 q 点的最短路径，但它需要转 3 个弯，而图中实线表示的是从 p 点到 q 点的最少转弯路径之一，它虽然路程长了一些，但只需要转 2 个弯，所以是本题的解。

如何才能保证搜索的结果是最少转弯路径而不是最短路径呢？他们在搜索的处理上关键的区别在什么地方呢？这是我们解题之前必须解决的问题。我们可以这样来思考：

求最短路径关心的是经过的方格共有多少，所以每一个拓展的方格都要计数，在一个出队结点的周围如果没有障碍物的话最多只有左、右、上、下相邻的 4 个结点属于同一层，他们具有一个共同的父辈结点（如图中 p 点周围的 a1、a2、a4、a6 属于同一层，其父辈结点都是 p），其他结点都不可能和它们同层。接下来这一层的每一个结点又可以向 4 个方向拓展（只要未拓展过且不出界、不是障碍物就行），……，直到找到目标结点为止。这时，其层次数减 1 就是路径的长度。由于广度优先搜索的原理保证了最先求得的解层次最少，所以就是最短路径。

求最少转弯路径呢？它关心的不是路径的长短而是转弯次数的多少，与搜索层次联系的不是路径长短而应该是转弯次数，也就是说不管路径多长只要没转弯就属于同一层次，就应该具有同一个父辈结点（如图中 p 点周围的 a1 至 a8 都属于同一层，其父辈结点都是 p）。搜索过程中转一次弯就必须深入一层，以保证记录下转弯次数，这又是如何做到的呢？我们只要做到沿着一个方向搜索到底，该拓展的结点全部拓展掉就行了，因为在这一方向上的所有结点如果要进一步拓展的话，在原来方向上已经无法拓展了，要么不能进一步拓展，要么就是在转过一个弯的方向上继续拓展，……，直到搜索到目标结点为止。这正是我们解题所需要的。

下面我们来分析一下本题的广度优先搜索的执行过程：

1. 从 p 点开始拓展：左面拓展了 a1 结点；右面拓展了 a2、a3 结点，遇障碍物停止该方向拓展；上方拓展了 a4、a5 结点；下方拓展了 a6、a7 a8 结点；（p 点是队列中的第 1 层，a 系列结点是队列中的第 2 层）
2. 下面对 a1 结点进行拓展：由于 p、a2、a3 结点位置都已拓展过，所以只能拓展 b1 至 b5 这 5 个位置的结点；（b 系列结点为第 3 层）
3. 同理：a2 结点拓展了 c1、c2 结点；a3 结点拓展了 d1 至 d5 结点；a4 结点拓展了 e1 至 e5 结点；……；（注意：只要是 a 系列结点拓展出来的结点都属于第 3 层）
4. 从图中可以看出搜索到目标结点的层次应该是第 4 层（可以从 e3 拓展得到）。那么我们分析一下转弯次数是多少呢？第 1 层是起点，第 2 层拓展的结点不转弯，第 3 层的结点转了 1 次弯，第 4 层的结点转了 2

次弯，……。所以转弯次数应是层次数减 2（题中设初始层为 0 层，所以转弯次数为层次数减 1）。

从上面的对搜索过程的分析中可以看出解决本题求最少转弯次数的关键是如何实现将同方向的结点一起进行拓展。为此，我们在程序中设置了 a、b 两个数组，a 数组记录障碍物的位置，b 数组动态记录搜索拓展过的位置。在循环处理时，如果是 a 数组表示有障碍物阻止或边界出界限制则终止该方向的搜索；如果是 b 数组表示该位置已经拓展过，只能做不拓展该结点的处理，但不能终止该方向上的继续搜索，因为跳过该结点在同一方向上还可能会存在未拓展过的结点，它们应属于同一拓展层次，所以不能漏掉（如图中 a4 结点拓展时跳过 c1、d1 结点继续拓展 e1 至 e5 结点那样）。

最后我们说明一下程序中用到的其它数组的作用：

常量数组 dx、dy 用于控制左、右、上、下 4 个搜索方向；

队列数组 d 设计了 3 个域，d[i,1]记录行坐标值，d[i,2]记录列坐标值，d[i,3]记录父辈结点地址；

c 数组用于搜索到目标结点后，在倒串搜索起点的过程中记录最少转弯路径的各转弯点坐标；

**参考程序：**

```
PROGRAM ex2_1_6(input, output,f);
CONST
  max=20;
  dx:array[1..4] of -1..1=(-1,1,0,0); {按左、右、上、下顺序搜索}
  dy:array[1..4] of -1..1=(0,0,-1,1);
VAR
  a,b            :array[1..max,1..max] of 0..1;
  c              :array[1..max*max] of byte;
  m,n,px,py,qx,qy:byte;
PROCEDURE inputdata;                        {从文件输入数据的过程}
  var
    i,j:byte;
    f  :text;
  begin
    assign(f,'data.txt');  reset(f);  readln(f,m,n);{读取行列值}
    for i:= 1 to m do     {读取方格平面图的数据。障碍物用 0 表示}
      begin
```

```
        for j:= 1 to n do  read(f, a[i,j]);  readln(f)
      end;
    readln(f,py,px);  readln(f,qy,qx);      {读取起点和终点坐标}
    close(f)
  end;
PROCEDURE BFS;                                  {广度优先搜索过程}
  var
    d        :array[1..max*max,1..3] of shortint;
    head,tail:0..max*max;
    i,j,k,x,y:byte;
  procedure start;        {广度优先搜索过程中嵌套的初始化子过程}
    begin
      d[1,1]:=py;  d[1,2]:=px;  d[1,3]:=0; {起点进队，父结点置 0}
      fillchar(b,sizeof(b),1);    {记录位置经过与否的数组置初值}
      b[py,px]:=0;     {从起点开始搜索，同时起点坐标记录数组置 0}
      head:=0;  tail:=1                      {队列首尾指针置初值}
    end;
  begin                                    {广度优先搜索过程开始}
    start;                                       {执行初始化过程}
    repeat
      inc(head);            {移动队首指针，准备队首结点出队拓展}
      for k:=head to tail do    {拓展前先检查队列中有无目标结点}
        if (qy=d[k,1]) and (qx=d[k,2]) then
          begin                             {找到目标结点后的处理}
            i:=k;  j:=0;        {从目标 k 结点开始倒串搜索至起点}
            while d[i,3]<>0 do          {c 数组用于记录解的路径}
              begin  inc(j);  c[j]:=i;  i:=d[i,3]  end;
            writeln('Total Across=',j-1);       {输出最少转弯数}
            write('(',py,',',px,')'); {从起点开始输出各转弯坐标}
            for i:=j downto 1 do
              write('-->(',d[c[i],1],',',d[c[i],2],')');
            writeln;  halt              {输出解后即终止程序运行}
          end;
      for k:=1 to 4 do              {按 4 个搜索方向进行结点拓展}
        begin
          y:=d[head,1]+dy[k];          {计算新拓展结点的位置坐
          x:=d[head,2]+dx[k];          标。y 表示行，x 表示列}
```

```
        while (x in [1..n]) and (y in [1..m]) and (a[y,x]=1) do
          begin   {新结点位置若在界内，也不是障碍物则进行拓展}
            if b[y,x]=1 then{同方向中的结点只要未搜索过即拓展}
              begin
                inc(tail);
                d[tail,1]:=y;  d[tail,2]:=x;       {记录坐标值}
                d[tail,3]:=head;              {记录父辈结点地址}
                b[y,x]:=0;       {已拓展的结点置 0 表示已搜索过}
              end;
            y:=y+dy[k];       {因为是搜索最少转弯路径，所以只要不转
            x:=x+dx[k]        弯就属于同一层，应沿一个方向搜索到底}
          end
      end;
  until head>=tail;                        {广度优先搜索终止条件}
  writeln('No Answer !')    {搜索终止还未找到解则输出无解信息}
end;
BEGIN   {主程序分为执行数据输入子程序和广度优先搜索子程序两部分}
  inputdata;  BFS
END.
```

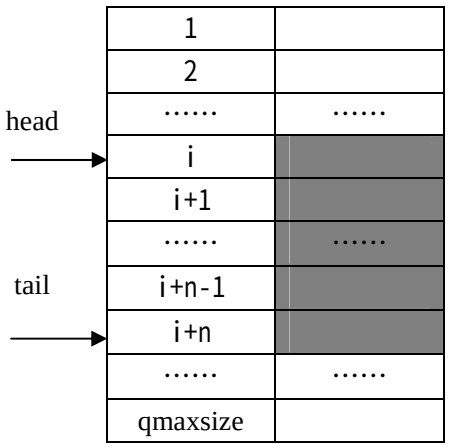
# 第二节　广度优先搜索的数据结构

本节我们来讨论一下广度优先搜索的数据结构类型。广度优先搜索是采用队列形式的数据结构来存放数据的。在广度优先搜索的问题中我们总是先建立一个数组形式的队列来存放已拓展的结点和待拓展的结点的，再在队列中设置了队首指针 head 和队尾指针 tail 来控制结点的进队和出队操作。为了对广度优先搜索能理解得更加透彻，我们应该对队列的数据结构知识有一个更进一步的全面的了解。

## 一、队列的概念：

队列(queue)是不同于栈的另一种线性表。对队列进行插入结点的操作只能限制在该线性表的一端进行，对队列进行删除结点的操作则限制在该线性表的另一端进行。在队列中可以进行插入操作的一端称为队尾，可以进行删除操作的一端称为队首。把一个结点元素插入队列中的操作叫做进队，从队列中删除一个结点元素的操作叫做出队。

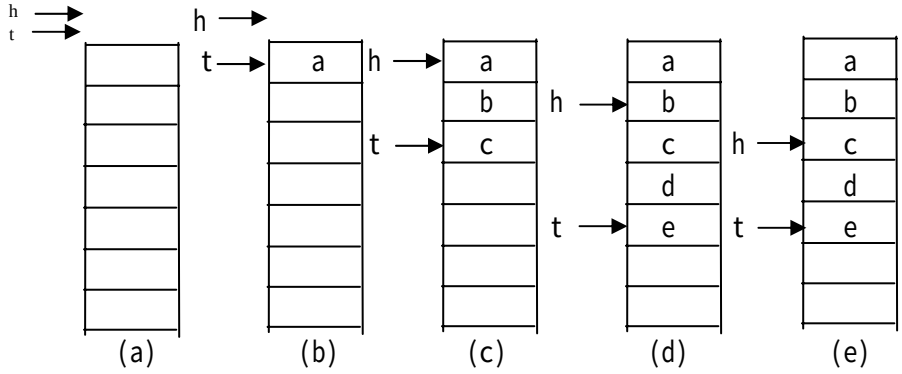
队列结构是从日常生活中抽象出来的。在日常生活中，售票窗外或服务台前，顾客按到达的先后次序排成一队。排队所遵循的原则是“先来先服务”，后来者总是加到队尾，排在队首的先到者总是首先取得服务，然后离队。所有的顾客一律平等，严格遵守秩序，不存在插队现象。也就是说队列符合以下规律：先放人队列中的元素首先取出，故队列又称为先进先出(FIFO：First In First Out)线性表。与堆栈相似,队列也可以采用顺序的方式存储数据。我们可以分配一块连续的存储空间（常采用一位数组的形式）来存放队列中的元素，再设置两个指针变量 head 和 tail 分别指向当前队列的队首元素和队尾元素。如图所示：




队列 q

图中所示的灰色背景部分表示是当前队列中的有效结点数据，队列的最大容量为qmaxsize,当前队首指针指着队列数组的第i个存储空间位置，队尾指针指着队列数组的第i + n个存储空间。

队列的基本操作如图所示：




(a)空队列。队首指针指向0，队尾指针指向0；

(b)第1个结点‘a’进队。队尾指针加1指向‘a’；

(c)结点‘a’出队拓展，结点‘b’,‘c’依次进队。队首指针指向‘a’(‘a’仍在数组中，但队首指针已下移，‘a’已不在队中)，队尾指针指向‘c’；

(d)结点‘b’出队拓展，结点‘d’,‘e’依次进队。队首指针指向‘b’(‘b’仍在数组中，但队首指针已下移，‘b’也已不在队中)，队尾指针指向‘e’；

(e)接下来将是结点‘c’出队……。

在队列的进队出队操作中，我们看到：图中每一个结点出队后(如‘a’结点)，后面跟着的结点(如‘b’、‘c’、‘d’等)并不依次往前移一位。跟旅客排队购票不一样：第一个刚离开，后面的就会依次跟上。其原因有两个：一是在计算机实现时，对于有n个结点元素的队列，前面的结点元素出队，其后的n - 1个结点元素要依次前移，即要进行n - 1次数组元素的交换，这样务必增加计算机完成一次队操作所需的时间；二是考虑到许多运用队列结构进行工作的算法，往往还会使用到已经出队的结点元素的数据。比如在广度优先搜索的许多问题中，我们还需要把新拓展出来的结点状态与前面曾经有过的所有结点状态进行比较，防止把相同状态的结点拓展进队而造成死循环，它要比较的结点对象不但是现在存在于队列中的对象，还要包括已经出队的数组中原来存在的所有对象。如果我们将出队的结点“真正”删除了，要么不能完成搜索任务，要么就得另外开辟一个数组来存放出队的结点。为了解决以上两个问题，我们可以让队列“固定”不动，而改为由服务员移动售票，让服务员顺着旅客队列移动，即通过不

断修改队首位置的方法来完成出队操作。

如采用移动“服务员”的思想，我们在队列中就要设置一个指向队首元素的指针 head（称为队首指针）和一个指向队尾元素的指针 tail（称为队尾指针）。当一个结点元素出队时，队首指针 head 就指向下一个位置，即 head=head+1；当一个结点元素进队时，队尾指针 tail 就指向下一个位置，即 tail=tail+1。注意：当 tail=pmaxsize 时新结点元素就不能进队了，因为队列已经满了。

## 二、队列的基本操作：

我们可以按照如下方式定义队列：

```
CONST
  qmaxsize = {队列的最大容量}；
TYPE
  qelement = {队列元素的类型}；
  queue= array[1..qmaxsize] of qelement；
  queueerror:(noerro，queueempty，queueunderflow，queueoverflow)；
VAR
  q        :queue；                                          {队列}
  tail,head:integer；                          {队尾指针和队首指针}
  qerro    :queueerror；
```

队列的操作常可定义成如下的函数和过程：

1. 判断队列是否为空的函数：

```
    FUNCTION qempty(var q:queue):boolean； {空队列则函数为 true}
      begin
        qempty:=(head >= tial)
      end；
```

2. 判断队列是否为满的函数：

```
    FUNCTION qfull(var q:queue):boolean；{队列已满则函数为 true}
      begin
        qfull:=(tail = qmaxsize)
      end；
```

3. 队列的初始化操作过程：　　{建立一个空队列}

```
    PROCEDURE qcreate(var q:queue)；
      begin
        head:=0；  tail:=0；
        qerror:=noerro
      end；
```

4. 结点的进队操作过程：　　{结点 x 进入队列 q}

```
PROCEDURE qinqueue(var q:queue；x: qelement)；
  begin
    if qfull(q)                          {判队列是否已满}
      then qerro:=queueoverflow          {如已满置上溢信息}
      else begin
             inc(tail)；                   {队尾指针加 1}
             q[tail]:=x                    {x 结点进队}
           end
  end；
```

5. 结点的出队操作过程：　　{将队首元素出队赋给 x}

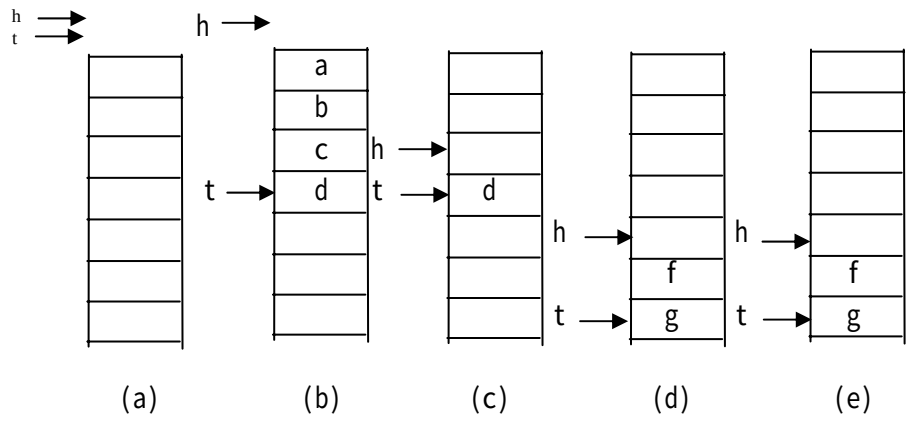
```
PROCEDURE qoutqueue(var q:queue；x: qelement)；
  begin
    if qempty(q)                         {判队列是否为空队列}
      then qerro:=queueunderflow         {如已空置下溢信息}
      else begin
             inc(head)；          {队首指针加 1，队首结点出队}
             x:= q[head]                 {x 取队首结点值}
           end
  end；
```

6. 取队首元素值的过程：{在队首元素不出队的情况下读取队首元素的值}

```
PROCEDURE qoutqueue(var q:queue；x: qelement)；
  begin
    if qempty(q)                         {判队列是否为空队列}
      then qerro:=queueunderflow         {如已空置下溢信息}
      else x:= q[head]                   {x 取队首结点值}
  end；
```

## 三、循环队列的概念：

由于队列只能在一端进行结点的插入操作，在另一端只能进行结点的删除操作，因此随着进队和出队操作的不断进行，就会出现一种和栈结构不同的情形：队列在存放结点的数组中不断地向队尾方向移动，而在队首的前面会产生一片越来越大的不能利用的空闲存储区，最后会导致当队尾指针指向数组最后一个位置（即 tail = qmaxsize）而不能再加入元素时，存储空间的前部却有一片存储区无端浪费，这种现象称为“ 假溢出 ”。下面我们用图示方法给出了一个“ 假溢出 ”的示例：




(a)空队列。队首指针指向 0，队尾指针指向 0；

(b)‘a’、‘b’、‘c’、‘d’四个结点进队，队尾指针加 4 指向‘d’；

(c)结点‘a’、‘b’、‘c’出队，队首指针指向最后出队的‘c’位置上，队尾指针仍指向‘d’；

(d)结点‘e’、‘f’、‘g’再依次进队，队尾指针指向‘g’(‘g’已经是数组的最大容量位置了)。这时，结点‘d’、‘e’又出队，队首指针指向‘e’出队前的位置；

(e)接下来再有结点要进队就会因为队尾指针 tail 已经指向数组的最大容量位置了而上溢出错。

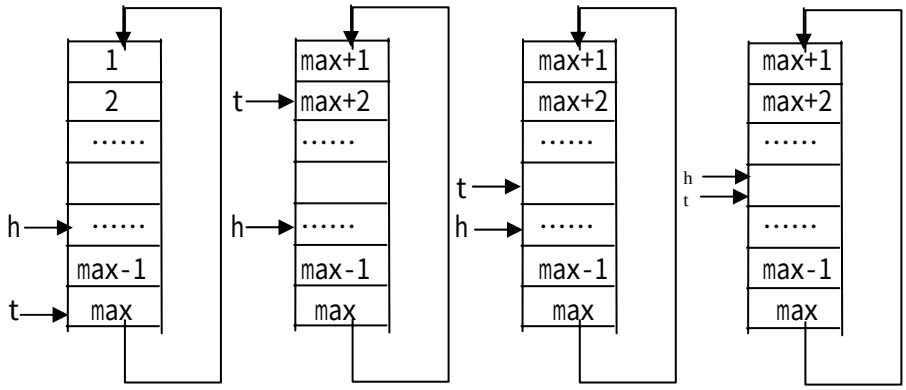
从图中我们看到虽然上溢出错，但实际的队列有效元素只有‘f’、‘g’两个结点，而‘f’前面的 5 个位置却空闲着未用，所以我们称其为“假溢出”。能否想办法把前面的空闲区域利用起来呢？回答是肯定的。为了利用起 head 前边空出来的可用空间位置，我们可以有这样两种处理方法：

一是采用队列移动的方法：当 tail=maxsize 时，检查队首指针 head 的前边是否有空位置剩余，若有 k 个剩余位置，则将队列中的所有元素全部前移 k 个位置，并将队首指针和队尾指针重新置位，head = head - k，tail = tail - k。

第二种方法是采用循环队列的方法：循环队列既可避免结点元素的移动，又能充分利用数组的空间位置。其优点是明显的，所以在处理很多队列问题时会在程序设计中经常使用。下面我们对循环队列的数据结构进行一下分析：

我们可以作这样的设想：在队列中，当存储空间的最后一个位置已被使用后还要进行进队操作时，只要存储空间第一个位置空闲，便可将进队元素加入到第一个位置，即将存储空间的第一个位置作为新的队尾。后面如有结点再要插入就顺着存储空间的第一个位置递增，而队首指针则仍然指向原来位置。一直到队尾指针“追上”队首指针时才是真正的队列满溢。

我们看到采用首尾相接的队列结构后，既可以有效地解决假溢出的问题，又避免了大量数据元素的移动，这就是所谓的环形队列（又称循环队列）。下面我们给出循环队列的结构示意图。




(a)队列“假溢出”，队尾指针指向数组最大空间值 max；

(b)再有 2 个结点进队，队尾指针循环绕到数组的下标 2 所指的位置，原来的队尾指针值 max+2 相当于 2；

(c)队列真正满的情况：队尾指针“追上”队首指针。队列满的条件是 head =(tail mod max)+1 ；

(d)队列空的情况：这时队首指针和队尾指针同指一个位置。队列空的条件是 head = tail；

循环队列将队列存储空间的最后一个位置绕到第一个位置，形成逻辑上的环状空间，供队列循环使用，循环队列结点的进出队列操作仍然遵循“先进先出”的队列数据结构的特点。

在环形队列中,队首指针 head 和队尾指针 tail 均可以环绕数组移动，这样数组中元素出队而腾出的空间即可以再利用，当 head 和 tail 的值大于数组最大容量值 qmaxsize 时，我们可以用求余运算（mod）使它们的值始终在[1..qmaxsize]之间，就象时钟的指针经过 12 点以后总是从 1 开始一样（如 13 mod l2 = 1)。

值得注意的是，为了方便处理和区分队满和队空两种状态，我们有意令队尾指针 tail 指向的位置与前面移动“服务员”队列情况稍稍不同，它指向队尾元素后面的一个空位置。并设定当 head = tail 时，队列为空；当 head =(tail mod qmaxsize)+1 时，队列为满。也就是说，为了方便区分队满和队空，我们是以牺牲了一个数组单元作为代价的。这样在队列中

我们虽然最多只能使用其中的 qmaxsize - 1 个元素了，但是程序的复杂程度却可以大大减少了。

对循环队列的操作我们归纳出以下几种状态：

① 初始时队列为空，队首指针和队尾指针均指向队列存储空间的最后一个位置，即 head = tail = qmaxsize。

② 执行进队操作时，队尾指针加 1，即

```
tail := tail+1;
if tail = qmaxsize + 1 then tail := 1;
```

这两条语句可用下面的一条语句替代：

```
tail := tail mod qmaxsize + 1;
```

③ 执行出队操作时，队首指针加 1，即

```
head := head + 1;
if head = qmaxsize + 1 then head := 1;
```

这两条语句也可用以下一条语句替代：

```
head := head mod m + 1;
```

④ 队列为空时队首指针和队尾指针相同，即 head = tail。

⑤ 队列为满时有 head = tail mod qmaxsize + 1。

## 四、循环队列的基本操作：

我们可以按照如下方式定义循环队列：

```
CONST
  qmaxsize = {循环队列的最大容量};
TYPE
  qelement = {循环队列元素的类型};
  queue= array[1..qmaxsize] of qelement;
  queueerror:(noerro, queueempty, queueunderflow, queueoverflow);
VAR
  q        :queue;                                    {循环队列}
  tail,head:integer;                          {队尾指针和队首指针}
  qerro    :queueerror;
```

循环队列的操作常可定义成如下的函数和过程：

1. 判断循环队列是否为空的函数：

```
FUNCTION qempty(var q:queue):boolean; {空队列则函数为 true}
  begin
    qempty:=(head = tial)
  end;
```

2. 判断循环队列是否为满的函数：

```
FUNCTION qfull(var q:queue):boolean;
  begin                          {若循环队列已满则函数值为 true}
    qfull:=((tail mod qmaxsize)+1 = head)
  end;
```

3. 循环队列的初始化操作过程：　　{建立一个空循环队列}

```
PROCEDURE qcreate(var q:queue);
  begin
    head:= qmaxsize;  tail:= qmaxsize;
    qerror:=noerro
  end;
```

4. 结点的进队操作过程：　　{结点 x 进入循环队列 q}

```
PROCEDURE qinqueue(var q:queue;x: qelement);
  begin
    if qfull(q)                          {判循环队列是否已满}
      then qerro:=queueoverflow            {如已满置上溢信息}
      else begin
             tail:=(tail mod qmaxsize)+1;     {队尾指针加 1}
             q[tail]:=x                         {x 结点进队}
           end
  end;
```

5. 结点的出队操作过程：　　{将队首元素出队赋给 x}

```
PROCEDURE qoutqueue(var q:queue;x: qelement);
  begin
    if qempty(q)                       {判循环队列是否为空队列}
      then qerro:=queueunderflow  {如循环队列为空置下溢信息}
      else begin
             head:=(head mod qmaxsize)+1;     {队首指针加 1}
             x:= q[head]       {x 取队首结点值,队首结点出队}
           end
  end;
```

6. 取队首元素值的过程：{在队首元素不出队的情况下读取队首元素的值}

```
PROCEDURE qoutqueue(var q:queue;x: qelement);
  begin
    if qempty(q)                       {判循环队列是否为空队列}
      then qerro:=queueunderflow  {如循环队列为空置下溢信息}
      else x:= q[head]                        {x 取队首结点值}
```

```
end；
```

## 五、队列结构在广度优先搜索中的应用：

队列的应用范围很广，其中最为典型的应用是广度优先搜索。广度优先搜索就是应用队列来实现的，在广度优先搜索中，我们将扩展出来的结点存贮在一个称作 q 的数组里，q 数组采用“先进先出”的队列结构。设两个指针 head 和 tail，分别是队首指针和队尾指针。 pnt 为父辈结点数组，它记录了每个结点的父辈结点地址，当搜索到目标后，可沿着父辈结点地址倒串回去，输出路径方案。

在广度优先搜索中，第一个搜索到的目标结点的路径，是搜索层次最少的路径，即最短路径。

广度优先搜索的基本算法过程如下：

① 初始化：

建立队列 q；设队首指针 head := 0； 队尾指针 tail := 1；

② 初始状态进队；队尾指针 tail := tail + 1；

③ repeat

```
  队首指针加 1；
  取 head 所指结点出队拓展；
  for i := 1 to maxrulenumber do     {在结点拓展规则范围循环}
    begin
      if 子结点符合拓展条件 then
        begin          {拓展条件是符合题意并且以前没有产生过}
          队尾指针加 1；
          新结点进队，存入队列的队尾；
          if 新结点即目标 then 输出并退出；
        end； {输出可设计子程序根据 pnt 数组数据倒串打印结果}
    end；
  until head >= tail；                    {队列空时终止搜索}
```

运用广度优先搜索算法解题要注意以下几个要点：

1. 广度优先搜索在搜索拓展产生新的子结点时，深度越小的结点越先得到拓展，即深度小的结点先拓展产生子结点。所以为了使广度优先搜索的算法便于实现，我们采用队列的数据结构来存放结点。
2. 无论问题性质如何不同，利用广度优先搜索来解题的基本算法是相同的。但对数组队列中每一结点的具体数据类型、产生的规则，则要根据不同的问题，采用不同的内容和结构。
3. 当结点到根结点的费用消耗和结点的深度成正比时，特别是当每一结

点到根结点的费用消耗等于深度时，用广度优先搜索首先得到的解即是问题的最优解。所以对要求解最有解的问题往往经常考虑采用广度优先搜索的算法。

4．广度优先搜索一般需存贮拓展产生的所有结点，占有的存贮空间要比深度优先搜索大得多。因此在程序设计中，必须考虑溢出和节省内存空间的问题。

5．广度优先搜索所需内存较多，一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索要快些，特别是在费用消耗与搜索深度成正比的情况下能较快地求出最优解。因此在对一个具体问题要决定究竟选择用哪一种算法时，要综合考虑，决定取舍。

6．广度优先搜索在拓展新结点时一般都要判断该结点是否曾经产生过，对已经出队的结点也要进行比较判断，所以一般不能采用循环队列的数据机构，否则还要另辟数组空间来存放已经出队的结点的内容。